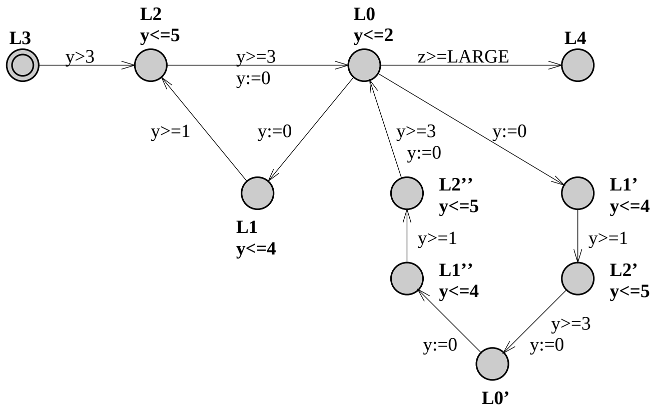
Development of Reactive Programs Using Uppaal



Martijn Hendriks

Development of Reactive Programs Using Uppaal

A master's thesis

Martijn Hendriks
Department of Computer Science, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Date:

February 14, 2002

Supervisors:

Dr. Jozef Hooman (University of Nijmegen)
Prof. dr. Frits Vaandrager (University of Nijmegen)
Prof. dr. Kim G. Larsen (University of Aalborg, Denmark)

Thesis number:

498

Acknowledgements

I thank Jozef Hooman, Frits Vaandrager and Kim Larsen for their enthusiastic and fine guidance. I also thank the people in Aalborg for the pleasant month, and in particular Carsten Stiborg for his help with the LEGO Production Cell.

Contents

1 Introduction **1**
1.1 Model checking and system development 2
1.2 Objectives 3
1.3 General outline 5

2 Setting and tools **7**
2.1 The hardware platform 7
2.1.1 The firmware and byte code 8
2.1.2 LegOS 9
2.2 The model checker 10
2.2.1 Timed automata 11
2.2.2 Symbolic model checking 13
2.2.3 The model checker Uppaal 14
2.3 Summary 15

3 Case studies: the conceptual designs **17**
3.1 A communication protocol 17
3.1.1 Encoding of packets on the RCX platform 18
3.1.2 Modeling the communication channels 18
3.1.3 The sender 19
3.1.4 The receiver 20
3.1.5 The tester 21
3.1.6 Validation of the protocol 22
3.2 The Level Crossing 22
3.2.1 The physical gate and the physical train 23
3.2.2 The gate controller 25
3.2.3 The train controller 27
3.2.4 The hurry dummy: urgent edges 27
3.2.5 Validation of the Level Crossing 27
3.3 The Production Cell 28
3.3.1 The physical robot arm 29
3.3.2 The brick template 30
3.3.3 The robot arm controller 31
3.3.4 The belt controller 34
3.3.5 Validation of the Production Cell 35
3.4 Summary 36

4 Compiling conceptual designs **39**
4.1 Extra information in conceptual designs 39
4.1.1 Only a Uppaal model is not enough 39
4.1.2 The type mappings 40
4.2 Definition of the translation 42
4.2.1 Generating the byte code program 42
4.2.2 Generating the Uppaal model of the run-time behavior 46
4.2.3 Relation between the input and output of the translation 49
4.2.4 Restrictions introduced by the translation 49
4.3 The implementation 51
4.3.1 Adding the type mappings to a Uppaal model 52
4.3.2 A small example 53
4.4 Experimental results 54
4.4.1 Testing the generated byte code of the Level Crossing 55
4.4.2 Testing the generated byte code of the Production Cell 57
4.4.3 Verifying the Level Crossing 58
4.4.4 Verifying the Production Cell 59
4.4.5 Discussion 60
4.5 Summary 61

5 Exact acceleration of real-time model checking **63**
5.1 Exact acceleration 65
5.1.1 An introduction to cycles 66
5.1.2 Acceleratable cycles 67
5.1.3 Acceleration 69
5.1.4 Experimental results 71
5.2 Acceleration of Uppaal models 72
5.2.1 Does equivalence of reachability disappear? 72
5.2.2 Automatic application of acceleration 76
5.2.3 Experimental results 78
5.3 Summary 80

6 Conclusion **83**

References **87**

Appendix A **91**

Chapter 1

Introduction

An *embedded real-time system* is a computer system that has been embedded in, and interacts with the environment and with the continuous elapse of time. These systems are called "reactive", since they must react to changes in the environment and to the passage of time. A typical example of an embedded real-time system is an airbag control system found in most newer cars. This system has to monitor the environment and inflate the airbag when the car crashes. In general, time plays a crucial role in these systems. It is not enough for the airbag to start inflating if the car crashes. An additional property should, e.g., be that the airbag has been inflated within 0.01 seconds after the start of the crash.

The importance of the correct functioning of many embedded real-time systems is clear. The name of a company may suffer severe damage if that company delivers an elevator system that ignores people waiting on the 13th floor. More dramatically, an error in the software for the airbag system could cost human lives. Besides being inconvenient or dangerous, errors in software are often very expensive to repair. Investigations show that most of the errors in software systems are introduced during the conceptual design phase and the programming phase [LRRA98]. In order to minimize the cost of repairing these, techniques to reveal errors must be integrated into the development process.

Validation and verification are techniques that ensure that software satisfies its specification and meets the needs of the software consumer. The difference between them is neatly summarized by Boehm [Boe79].

Validation: "Are we building the right product?"

Verification: "Are we building the product right?"

Both validation and verification are needed in the software development process. A program that meets its specification, but that is not useful to the customer is as bad as a program that is useful, but that sometimes displays unwanted behavior.

Nowadays, testing is the most widely used technique for validation and verification [Som96]. This involves exercising the realized system using data like the real data processed by the system. Another important technique is peer reviewing or program inspection [Fag86]. This is a completely manual activity in which the design is reviewed by a team of developers that have not been involved in the design of that particular part. Both methods have clear disadvantages. Since testing can – in general – only cover a small subset of the possible system behaviour, it can only show the presence of an error and not its absence. Moreover, this technique needs executable code. Finding an error means rewriting the code and new tests, which both are very time consuming. It is clear that peer reviewing also is very time consuming and error prone due to the "human factor" and the increasing complexity of modern designs. It seems that these techniques are only useful for validation purposes. As for verification, one cannot rely on testing and peer reviewing.

Formal methods, like model checking and formal verification are already successfully applied in the area of hardware and protocol verification. However, their use in software engineering is still limited. In this project we focus on the use and integration of model checking – as a means to validation *and* verification – in the software development process of reactive programs for embedded real-time systems.

1.1 Model checking and system development

Among others, model checking has emerged as a practical tool for the validation and verification of systems [CK96]. In this approach, the system is modeled as a (possibly) infinite state machine. The state space of this model can be (symbolically) explored to discover whether or not the model satisfies some given specification.

Nowadays, model checking might typically be applied at two different phases of system development. First, model checking can be used to detect conceptual design errors in an early stage of system development. A model of the system and its environment can be constructed to obtain confidence in the correctness of the conceptual design. Since only the conceptual design is modeled, no assumptions about the real run-time behaviour should be made based on this model. Thus, the correctness of the implementation cannot be assured. Typical examples include the verification of the design of a gear box controller [LPY].

Second, a model of the run-time behaviour of the system can be constructed during or after the implementation. Typically, all hardware details and parameters must be present in this model. One disadvantage of this approach is that these models tend to become very large and difficult to verify. Another disadvantage is that this approach must be applied *after* (part of) the development process since it needs executable code. Discovering an error means that (part of) the development process must be passed through again. This is, of course, a time and money consuming business. The Bandera project aims at a general and automated realization of this approach. It provides tools to verify Java source code by the extraction of finite-state models that can be feeded into various model checkers [CDH$^+$00]. Another example is the translation from LEGO RCX byte code to a UPPAAL model [IKL+00]. Using this translation, one can model-check the real run-time behaviour of the programs running on the RCX.

1.2 Objectives

In section 1.1 we described two separate phases of the development process in which model checking might be applied. In general, no formal relation exists between the models in these separate phases. By establishing this relation, some of the mentioned disadvantages can be avoided. In our approach, the relation consists of a translation from the models that occur in the conceptual design phase to executable code. However, this translation cannot preserve all properties of the conceptual design. For example, a conceptual design – in general – abstracts from the hardware on which it is to be implemented. As a result, it is very likely that properties concerning timing and duration of events are different in the actual executable code. To support the verification of the run-time behaviour of the generated executable code, we also generate a model of this code. Figure 1.1 depicts this approach. The oval boxes contain the models that can be used for val-

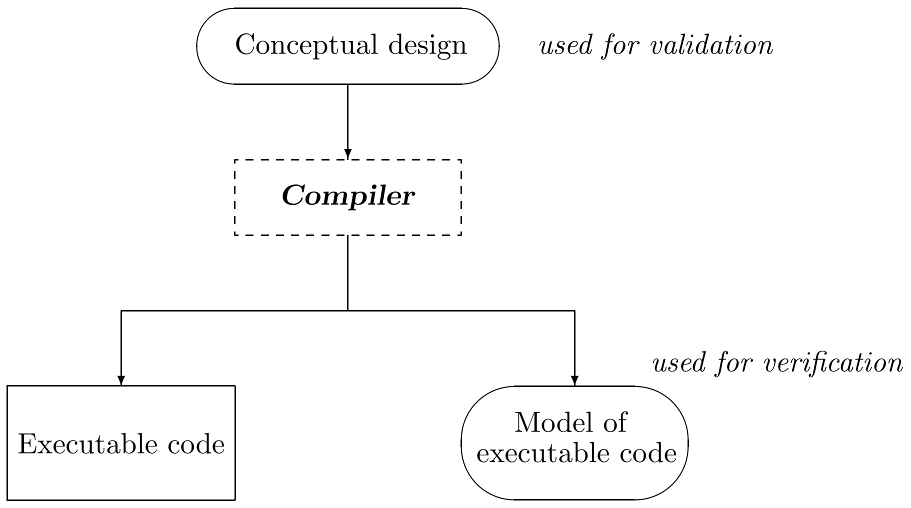


Figure 1.1: The proposed technique.

idation and verification purposes using the associated model checker. The dashed box contains the translation. Such a translation has the following advantages:

- It enables the *automated generation* of executable code and a model of the executable code from the conceptual design. The automated process saves implementation time and it minimizes the risk of errors during the implementation.

- The system developer can concentrate on the conceptual design. The exact relation between the conceptual design and the implementation is provided by the translation.

Despite these advantages, the practical benefits of such a translation are not clear. The translation might be very restrictive in the sense that only a very limited subset of the modeling language can be translated. Moreover, the translation is very dependent on the target platform of the system. It is not clear whether or not a translation can be constructed for every platform. We investigate the practical benefits of the translation in one particular setting. The main question we seek an answer to is the following.

> *What are the benefits of a translation from models appearing in the conceptual design phase to (i) executable code and (ii) models of that code, for the development of reactive programs for embedded real-time systems?*

One of the disadvantages mentioned in section 1.1 is that the models of the executable code are in general hard to verify due to their large state spaces. This is a significant problem for the generality and scalability of our approach. Thus, a second question we seek an answer to is the following.

> *How can we accelerate the model checking process of the model of the executable code?*

Possibly, a translation enables the automatic generation of "intermediate" models, see figure 1.2. These are of a complexity between the model of

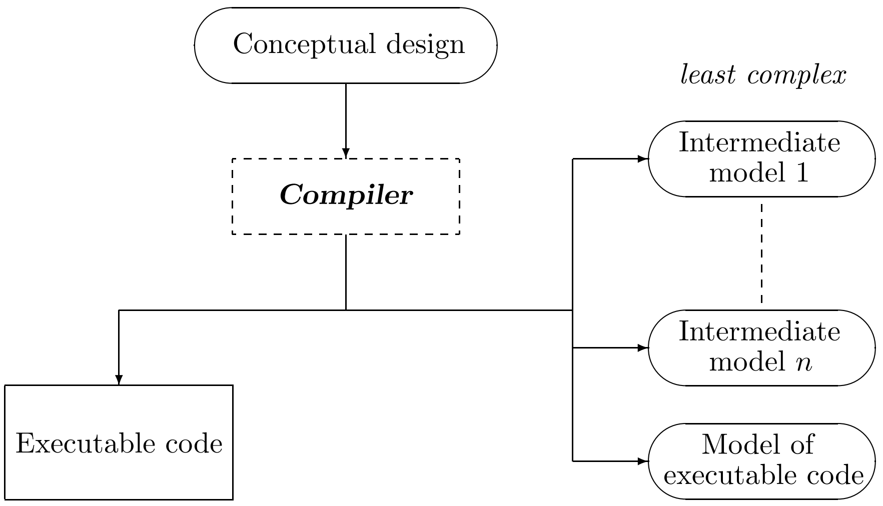


Figure 1.2: Extending the technique.

the conceptual design and the model of the executable code. These models can then be used for certain verification questions and – hopefully – model checking is accelerated with these intermediate models. Concluding, our approach to obtain correct reactive programs consists of three steps:

(1) Construction of a model of the conceptual design of the system. The validation of the design can be obtained using this model.

(2) Compilation of the model of step (1) to executable code and models of the run-time behaviour of this code.

(3) Verification of the executable code using the models obtained in the second step.

In this thesis, we construct the techniques to implement this approach for a fixed model checker and hardware platform. We apply the approach to two case-studies in an attempt to answer the two questions stated above.

1.3 General outline

In this thesis we implement a technique to obtain correct reactive programs. In short, this technique consists of a translation from conceptual designs to (i) executable code and (ii) a model of the run-time behaviour of that code. The main objective is to answer two questions concerning our technique. First, we would like to know its benefits and drawbacks. Second, we investigate the possibilities of accelerating the model checking process of the generated models, since these tend to become very large.

The first logical step is to choose a testing environment. In chapter 2 we fix and motivate the hardware platform and the model checker. We summarize the main features of the hardware platform and the language that we use to program it (which thus is the language of the executable code that we generate). Moreover, we discuss the theoretical background of the model checker, which proves to be necessary for our investigation to the possibilities of acceleration of the model checking process.

The starting-point of our technique is a conceptual design of a reactive system. In chapter 3 we show and explain the conceptual designs of two reactive systems: the *Level Crossing* and the *Production Cell*. These are designs for existing hardware, which is required for the testing of the generated executable code. Of course, we validate our designs using the model checker.

The core of our technique is the translation from a conceptual design to executable code and a model of the run-time behavior of that code. We develop and explain the concepts of this translation in chapter 4. We implement this translation and we illustrate its concepts and operation by applying it to a small example. Moreover, we apply our compiler to the conceptual designs of the Level Crossing and the Production Cell. We test the resulting executable code on the physical systems, and we *try* to model check the executable code, using the generated models.

In chapter 5 we develop theory to accelerate the model checking process for a certain class of models in an *exact* way. We illustrate our theory of exact acceleration with experimental results of a theoretical example. These results very nicely demonstrate the "mechanics" of a main result, which says that our exact acceleration technique actually works.

Unfortunately, generalization of our technique to a broader class of models, which contains the Level Crossing and the Production Cell, seems not possible without loss of exactness. Instead, our technique becomes an *over-approximation*, which can still be used to verify the truth of certain types of properties. It is easy to apply our technique automatically to the models of the generated executable code, and we add this feature to the compiler. Thus, we generate one "intermediate model" as in figure 1.2. Moreover, we apply our over-approximating acceleration technique to the Level Crossing and to the Production Cell, and we show the improved performance of the model checking process.

The last chapter of this thesis, chapter 6, contains the summary and overall conclusions. We answer the two questions concerning our technique, and we state possibilities for future work.

Finally, we note that all material that appears in this thesis – the models and the implemented tools – is available at the web site of this thesis, which can be found at
`http://www.cs.kun.nl/ita/publications/mastertheses/200202_martijn_hendriks`.

Chapter 2

Setting and tools

In the introduction we explained the technique that we use to develop reactive programs that function correctly (see figure 1.2 on page 4). The first part of this technique consists of the construction and validation of a conceptual design using some model checker. The second step is to generate executable code and one or more models of the run-time behavior of that code. These generated models are used for the verification of the real executable code.

To answer the main question stated in the introduction, we implement the aforementioned translation for a fixed model checker and hardware platform. In section 2.1 we introduce the LEGO RCX hardware platform that is used and we motivate this choice. In section 2.2 we explain why the model checker Uppaal is very well suited to our end. Finally, we explain the theoretical background – timed automata – of this model checker.

2.1 The hardware platform

We consider a special class of embedded real-time systems, namely those controlled by micro controllers. These are computation devices consisting of a CPU, ROM and RAM and an A/D converter with I/O ports. All these subsystems are integrated in order to make development easy and to achieve a low selling price. More specifically, we use the LEGO RCX brick (or RCX for short) as the control center of our embedded real-time systems [Nie00]. This big LEGO brick has an Hitachi H8/3292 micro controller inside that runs with a clock speed of 16 MHz. The 16 kB of ROM on the CPU has been extended with 32 kB of external RAM.

The RCX has the possibility to use three sensors, e.g., light sensors and touch sensors, and three actuators, e.g., motors and lights. Moreover, it possesses an infrared port that can be used for communication with other RCX bricks and for downloading programs to the RCX. Finally, the RCX has a notion of time for it has two timer modules. We use this hardware platform for the following reasons.

- Availability. This “toy” is widely available. In particular, it is available at the university of Nijmegen and at the university of Aalborg, which was visited during this project.

- Flexibility. The LEGO system enables the construction of various different embedded systems in an easy way. Thus, case-studies are easy to come up with.
- Realism. The hardware specifications of the RCXs micro controller are very similar to those encountered in the real live, e.g. electronics in washing machines, cars and microwave ovens.
- Simplicity. This toy is easy to use.

The RCX is standard equipped with firmware that contains a byte code interpreter, and consequently the programs for the RCX must be written in the byte code language. Normally, RCX programs are developed on a personal computer. When finished, the programs can be transferred to the RCX using an infrared tower that is connected to a serial port of the personal computer. Running of the programs is easily done by pressing a button on the RCX brick.

2.1.1 The firmware and byte code

This section gives a brief introduction to the firmware of the RCX; for detailed information about the internals of the RCX, its firmware and the byte code language, we refer to the web site *RCX internals* [Pro99].

The firmware with which the RCX is standard equipped contains an interpreter for programs written in a byte code language. It leaves only 6 kB of memory for user programs. These programs consist of at least one and at most ten *tasks* which can use up to eight *subroutines*. Each program can use 32 16-bit *regular* variables for computation purposes. The firmware employs a simple round robin scheduler to interleave the active tasks of a running program; each task may execute one byte code instruction before control is transferred to the next active task.

The byte code language contains approximately 100 instructions that can be divided into "management" instructions for communication between the personal computer and the RCX, and into "normal" instructions for the actual programs. The language offers an extended set of instructions for conditional branching, manipulation of sensors and actuators, and manipulation of regular variables.

As an example, consider figure 2.1 which contains a very simple program for the RCX in pseudo code. It consists of one task that uses one regular variable `a`. The while loop increments the value of `a` until it is equal to or greater than 100. Figure 2.2 shows the three symbolic byte code instructions (on the left) and the actual array of bytes (on the right) that implement this simple program. The first instruction is a "branch always far" instruction that transfers control to instruction at address 8. This third instruction is a "test and branch far" instruction that tests whether or not `a<100`. If not, then control is transfered to the instruction with address 3. This instruction just adds 1 to variable `a`. Otherwise, the program halts.

```
task main
{
    int a;
    while(a<100)
        a=a+1;
}
```

Figure 2.1: Small program in pseudo code.

```
000 baf 8                72 07 00
003 add v[0], 1          24 00 02 01 00
008 tbf 99>=var[0], 3    95 42 00 63 00 00 f5 ff
```

Figure 2.2: The equivalent symbolic byte code (left) and the actual byte code (right).

There exists an alternative for the standard firmware, being LegOS. In the next subsection we explain why we do not use this more advanced operating system.

2.1.2 LegOS

LegOS implements a real embedded operation system featuring pre-emptive scheduling and dynamic loading of programs [Nie00]. Using this operating system instead of the standard firmware has the following advantages:

- There is more memory available for user programs.
- Since this is a real OS and not an interpreter, the executable code is the native machine language of the micro controller. An adapted version of the GNU C cross-compiler exists such that programs can be written in plain C. Typically, these programs run 5-10 times faster than byte code programs for the interpreter.
- Using LegOS and C is slightly more expressive than the byte code and the interpreter.

Despite all these advantages we do not use LegOS. In the introduction we explained that we automatically generate executable code *and* an exact model of that code. Since this model is intended for the final verification of the system, it must be very accurate. Using LegOS means that the model should cover the complete program with the single native instructions, these are the atomic units of execution for the scheduler, as building elements. Moreover, the relevant parts of the OS – like the system interrupts, the scheduler and other kernel tasks – must also be modeled, including timing parameters, as they significantly direct the execution of the program.

In comparison to the very detailed and low-level model that should be used for the LegOS, the model for byte code programs for the original firmware is fairly simple. Moreover, such a model, including timing parameters, already exists and – we think – is accurate enough [IKL$^+$00]. The difference in need for detail is made by the virtual machine of the firmware. It "shields" the program from the real hardware and the byte code instruction is the atomic unit of execution.

Concluding, the models of the executable code of LegOS programs will probably be much more complicated than those of byte code programs. This increased complexity decreases the chances for systems to be practically verifiable, which is a main feature of our technique.

2.2 The model checker

To design the reactive programs for the RCX, we use the timed automata of Alur and Dill [AD90]. Several case-studies have shown that these timed automata are suitable for modeling a large class of real-time systems. Since we only want to develop relatively small and simple programs, the expressivity of timed automata probably suffices to use them as a design tool. However, if our approach is applied in a more complicated (realistic) setting, then the need for more powerful design tools arises.

A more general class of systems can be modeled using linear hybrid automata, or the equivalent stopwatch automata [Hen96, CL00]. However, the reachability question for hybrid and stopwatch automata is undecidable [HKPV98]. Still, a model checker for linear hybrid systems has been implemented, namely HyTech [HHWT97]. Due to the undecidability, the model checking procedure used in this tool may not terminate.

Fortunately, the reachability question and, more general, the question whether or not a TCTL formula is satisfied, are decidable for timed automata and various model checkers exist, e.g. Kronos and Uppaal [ACD93, Yov97, LPY98]. We use the model checker Uppaal for the following reasons.

- Uppaal uses only a subset of the logic that Kronos uses. However, Uppaal can use more efficient algorithms to explore the state space. In this project, the efficiency of model checking is the most important of the two.

- Uppaal can use bounded integer variables and Kronos cannot use integer variables at all. This feature is very convenient and almost necessary for the modeling of larger systems.

- Uppaal has a nice graphical user interface, whereas Kronos uses a "textual" interface. Therefore, Uppaal is easier to use.

- Finally, there is a historical argument: Uppaal is often used in Nijmegen and we were already familiar with this tool. Moreover, it is developed in Aalborg, which has been visited for this thesis.

In the next subsections we explain the theoretical background of Uppaal. We first summarize a theory of timed automata, then we focus on the symbolic model checking techniques for timed automata, and finally we briefly introduce Uppaal.

2.2.1 Timed automata

This section has been based on the work of Alur and Dill [AD90, Alu99]. In order to define finite automata that use real valued clocks, the set of clock constraints over a set of clock variables is defined. Let X be a set of clock variables, then the set $\Phi(X)$ of clock constraints ϕ is defined by the following grammar, where $x \in X$, $c \in \mathbb{N}$, and $\sim$ denotes one of the binary relations $<, \leq, =, \geq$ or $>$.

$$\phi := \mathit{true} \,|\, x \sim c \,|\, \phi_1 \wedge \phi_2$$

A *clock interpretation* ν for a set X is a mapping from X to $\mathbb{R}^+$, where $\mathbb{R}^+$ denotes the set of nonnegative real numbers. A clock interpretation ν for X satisfies a clock constraint ϕ over X, denoted by $\nu \models \phi$, if and only if ϕ evaluates to *true* with the values for the clocks given by ν. For $\delta \in \mathbb{R}^+$, $\nu + \delta$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + \delta$. For a set $Y \subseteq X$, $\nu[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$ and agrees with ν over the rest of the clocks. We let $\Gamma(X)$ denote the set of all clock interpretations for X.

Definition 2.1 (Timed automaton) *The tuple $(L, l^0, \Sigma, X, I, E)$ defines a timed automaton, where*

- *L is a finite set of locations,*
- *$l^0 \in L$ is the initial location,*
- *Σ is a finite set of labels,*
- *X is a finite set of clocks,*
- *I is a mapping that labels each location $l \in L$ with some clock constraint in $\Phi(X)$ and*
- *$E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of edges. An edge $(l, a, \phi, \lambda, l')$ represents a transition from location l to location l' on the symbol a. The clock constraint ϕ specifies when the edge is enabled and the set $\lambda \subseteq X$ gives the clocks to be reset with this edge.*

Example 2.1 *Consider the timed automaton depicted in figure 2.3. The locations are depicted as vertices labeled with the name of the location. Location* `L3` *is the initial location. The set of labels consists only of the* empty *label τ, that is not depicted. The clocks are* `y` *and* `z`*. The invariant mapping I is given by the bold clock constraints depicted at the locations. If a location has no clock constraint associated with it, then we assume that the invariant for that location is* ***true****. The edges may be labeled with a clock constraint and a set of clock resets. If an edge is not labeled with a clock constraint, then the guard of this edge is* ***true****. If an edge is not labeled with a set of clocks, then no clocks are reset on the edge, i.e. $\lambda = \emptyset$ for that edge. Finally, the object* `LARGE` *that appears in a clock guard on the edge from location* `L0` *to* `L4` *is a constant natural number.*

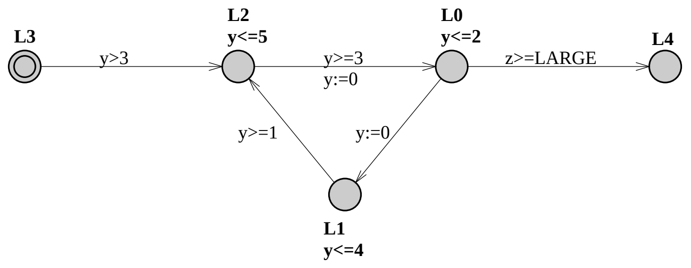


Figure 2.3: Timed automaton P.

The semantics of a timed automaton A is defined by associating a transition system S_A with it. A state of S_A is a pair (l, ν), where l is a location of A and ν is a clock interpretation for X such that ν satisfies $I(l)$. There are two types of transitions in S_A:

- Let $\delta \in \mathbb{R}^+$. We say $((l, \nu), (l', \nu'))$ is a δ-*delay transition*, iff $l = l'$ and $\nu' = \nu + \delta$.
- Let $a \in \Sigma$. We say $((l, \nu), (l', \nu'))$ is an a-*action transition*, iff an edge $(l, a, \phi, \lambda, l')$ exists such that $\nu \models \phi$ and $\nu' = \nu[\lambda := 0]$.

Using this transition system, we can define the traces of a timed automaton.

Definition 2.2 ((l, ν)-trace) *Let $M = (L, l^0, \Sigma, X, I, E)$ be a timed automaton. We say that a finite or infinite sequence $((l_0, \nu_0), (l_1, \nu_1), ...)$ is a* (l, ν)-trace *of M, if*

- *$l_0 = l$ and $\nu_0 = \nu$, and*
- *$((l_i, \nu_i), (l_{i+1}, \nu_{i+1}))$ is an a-action transition for some $a \in \Sigma$ or a δ-delay transition for some $\delta \in \mathbb{R}^+$, for all $i \geq 0$ that appear in the sequence.*

We call a (l, ν)-trace *compressed*, if that (l, ν)-trace starts with a delay transition, and it does not contain two consecutive action or delay transitions. Thus, after every action transition follows a delay transition and after every delay transition follows an action transition. Finally, we let $Tr(M)$ denote the set of all (l^0, ν_{init})-traces of a timed automaton M, where l^0 is the initial location of M and $\nu_{init}(x) = 0$ for all clocks x of M.

We are primarily concerned with reachability properties of timed automata. A reachability property ϕ of a timed automaton M is of the form $\exists\Diamond(P)$, where P is a *state* property of M. The state properties P are interpreted over the states of M, whereas the reachability properties ϕ are interpreted over M. For example, a reachability property for automaton P of example 2.1 is the following:

$$\exists\Diamond(\texttt{L2} \wedge (\texttt{y<=4} \vee \texttt{z>20}))$$

Informally this property means that a state (l, ν), such that $l = \texttt{L2}$, and $\nu(\texttt{y}) \leq 4$ or $\nu(\texttt{z}) > 20$, is reachable. This example also demonstrates the straightforward satisfaction relation that is used to interpret state formulas over the states of a timed automaton.

Using the traces of a timed automaton and the satisfaction relation for the state properties, we can define the satisfaction relation for the reachability properties.

Definition 2.3 (Reachability) *For a timed automaton M and a property $\phi = \exists\Diamond(P)$, we say that M satisfies ϕ, denoted by $M \models \phi$, if a trace $((l_0, \nu_0), (l_1, \nu_1), ...) \in Tr(M)$ exists, such that $(l_i, \nu_i) \models P$ for some $i \geq 0$.*

Note that we can express safety or invariance properties with reachability. The state property P is an invariant – P always holds – if and only if $\neg\exists\Diamond(\neg P)$. We will use the abbreviation $\forall\Box(P)$ for such invariance properties.

In the next section we will explain forward symbolic reachability analysis that is used by UPPAAL and KRONOS.

2.2.2 Symbolic model checking

The transition system defined by a timed automaton has uncountable many states. This renders the straightforward application of traditional finite-state model-checking algorithms impossible. However, Alur et al introduced the so-called regions as a finite-state symbolic technique for proving decidability of reachability as well as model-checking for TCTL [ACD93].

Unfortunately, the number of regions grows exponentially in the size of the constants used in the model, and are therefore not particularly useful when constructing efficient model-checking tools. Instead, real-time model-checkers such as UPPAAL and KRONOS are based on so-called zones, which provide a representation of convex sets of clock interpretations as constraints on (lower and upper) bounds on individual clocks and clock differences. As an example consider the set of clock interpretations for the two clocks x and y described by the following constraints:

$$\begin{aligned}
&0 \leq x \leq 5\\
&7 \leq y \leq 12\\
&y - x = 7
\end{aligned}$$

Zones may efficiently be represented as *Difference Bounded Matrices* [Bel57, Dil89], which offers a canonical representation for constraint systems. Furthermore, the canonical form allows inclusion-check as well as the effect of action and delay transitions to be computed efficiently (i.e. time complexity mostly quadratic and in worst case cubic in the number of clocks)

The model-checking engine of UPPAAL performs a symbolic exploration of the reachable symbolic state space on-the-fly, starting with an initial state. For each unexplored symbolic state, its successors due to delay and action transitions are computed, and compared to already explored states. If a successor has already been seen in the past, it is discarded. On the other hand, if a successor has not yet been seen, it is added to the list of states waiting to be further explored.

To illustrate forward symbolic exploration consider the timed automaton of figure 2.3 on page 12. Depending on the value of `LARGE`, the cycle in automaton P must be executed a certain (large) number of times before the edge to location `L4` is enabled. In table 2.1 we show the symbolic states that result from one execution of the cycle starting in the initial state. This

State #	Location	Zone		
1	`L3`	$y = 0$	$z = 0$	$z - y = 0$
2	`L2`	$3 < y \leq 5$	$3 < z \leq 5$	$z - y = 0$
3	`L0`	$0 \leq y \leq 2$	$3 < z \leq 7$	$3 < z - y \leq 5$
4	`L1`	$0 \leq y \leq 4$	$3 < z \leq 11$	$3 < z - y \leq 7$
5	`L2`	$1 \leq y \leq 5$	$4 < z \leq 12$	$3 < z - y \leq 7$
6	`L0`	$0 \leq y \leq 2$	$6 < z \leq 14$	$6 < z - y \leq 12$

Table 2.1: Simulation data of P.

simulation data shows that the sixth symbolic state is not contained in the third, since the zone of the sixth symbolic state contains clock interpretations that are not in the zone of the third symbolic state, and therefore this state is explored further. In general, every new execution of the cycle gives rise to new symbolic states.

2.2.3 The model checker UPPAAL

The tool UPPAAL is a model checker for networks of timed automata that can use (arrays of) bounded integer variables and binary synchronizations. A small subset of the temporal logic TCTL can be used for specification. In this section we briefly introduce UPPAAL; for a detailed description we refer to the paper "UPPAAL in a Nutshell" [LPY98] and to the web site `http://www.uppaal.com/`.

An UPPAAL model consists of a bounded number of concurrent *processes*, each of which is described by a timed automaton. Each process may have local clocks and variables, that can only be used by itself, and it can use all global clocks and variables. The processes can communicate by the global variables and by binary synchronizations.

As with timed automata, the semantics of an UPPAAL model is defined by the underlying transition system. A state in the transition system is a vector $(\vec{l}, \nu, v)$, where $\vec{l}$ is a vector that contains the current location for each process, ν is a clock interpretation and v is a *variable interpretation.* We do not explicitly define the transition system here, since it is very similar to the transition system of regular timed automata. However, we explain the largest differences:

- Blocking binary synchronizations. The synchronization labels on edges are τ or $(b, !)$ or $(b, ?)$, where b is a label or *channel.* In the first case, the edge does not synchronize and it is executed alone. In the last cases, the edge must synchronize with a matching edge of another process and the action transition thus involves both edges (a ? side matches with a ! side). The assignments on the ! side of a synchronization are executed first.
- Urgency of synchronizations and locations. If some channel a is declared as *urgent*, then if an action transition is enabled that synchronizes over a, then no delay transition may take place. The same holds for the situation where the location vector contains a location l that has been declared urgent.
- Commitment of locations. If the location vector contains a location l that has been declared *committed*, then no delay may take place and the next action transition *must* involve a process that is in a committed location.

The model checking engine of UPPAAL version 3.2 can not only check reachability – and thereby invariance – properties as in definition 2.3 on page 13, but also the TCTL formulas $\forall\Diamond(P)$ and $\exists\Box(P)$, where P is a state property. Informally, the first property means that for all traces P will eventually hold. The second property means that there exists a trace on which P always holds. There also exists a special predicate `deadlock` that can be used to detect situations in which no action transition is enabled.

The model checking engine of UPPAAL has some useful features. First, it is possible to choose the search order: breadth-first (default) or depth-first. Second, it supports the so-called *convex hull* approximation, which is an over-approximation of the symbolic state space. Typically, explorations using this approximation are finished faster than normal explorations. However, only the truth of invariance and the untruth of reachability properties can be verified, since it is an over-approximation.

2.3 Summary

This chapter introduced the setting in which we implement and test our technique for obtaining correct reactive programs by using a model checker in the development process. Recalling figure 1.2 on page 4, we use a model checker to construct a conceptual design, automatically translate this design to executable code and one or more models of the run-time behavior of the code. The model of the conceptual design is for validation purposes and the other models are intended for verification purposes.

In this chapter we decided to use the LEGO RCX as target platform of the translation for its availability, flexibility and – last but not least – its realism. We use the formalism of timed automata, in the shape of the model checker UPPAAL, for the conceptual modeling and for the modeling of the run-time behavior of the system.

The next chapter describes two case-studies that we use to evaluate our technique. We explain the *Level Crossing* and the *Production Cell* and give the conceptual designs of these systems.

Chapter 3

Case studies: the conceptual designs

This chapter contains two case-studies that we use to evaluate our technique for obtaining correct reactive programs for the RCX using the model checker UPPAAL.

In section 3.1 we discuss our conceptual design of a communication protocol. We need this protocol to implement reliable communication between RCXs, since the standard firmware of the RCX only offers unreliable communication primitives. Both case studies – the Production Cell and the Level Crossing – use this protocol.

Section 3.2 describes the conceptual design of a LEGO Level Crossing with two RCXs. Finally, section 3.3 contains the conceptual design of a LEGO Production Cell, also consisting of two RCXs. For all three systems, we state and check properties in order to validate the conceptual designs.

The descriptions of the UPPAAL models are not complete since we only explain the concepts and omit the details. For the exact models we refer to the web site of this thesis.

3.1 A communication protocol

Our communication protocol is based on the alternating bit protocol that provides reliable data transfer over an unreliable channel, such as the air between two RCX bricks [Tan96]. The main goal of the protocol is to provide an ordered data-stream from a sender to a receiver. The two main problems are that packets may get lost and must be retransmitted by the sender, and that the order of the packets must be preserved. The detection of the loss of a packet is achieved by a time-out strategy, while preservation of the order is achieved by the use of an alternating bit attached to each packet.

The ultimate goal of our model of the protocol is to implement it on the RCX platform. Therefore, the next section describes how we can encode the packets, including their alternating bits, in such a way that the RCX can support their sending. Next, we describe how we modeled the lossy channel between the RCXs, the sender and the receiver of the protocol. Finally, we describe a *test* process that provides input to the sender and that reads the output of the receiver. This process is used for validation purposes only.

3.1.1 Encoding of packets on the RCX platform

We modeled the protocol with the intent to implement it on the RCX platform, which supports an unreliable infrared communication channel. The byte code language offers an instruction to broadcast a number between 0 and 255 to all other RCX bricks in range. A natural choice is to let these numbers be the packets. However, each packet must have an "alternating" bit attached to it. It is clear that the only way to achieve this, is to *encode* this bit in the number that is broadcasted.

Each packet is a number between 0 and 125. This enables us to encode the alternating bit in the following way: $r = 2p + b + 3$, where r is the real value transmitted by the RCX hardware, p is the packet and b is the alternating bit. We can decode the bit and the packet from the real value as follows: $b = 1 - (r \ mod\ 2)$ and $p = (r - b - 3)/2$.

This scheme does not make use of the real values 0 to 2 and 255. The reason for this is that the receiver needs to acknowledge the reception of a packet. This acknowledgement must include the bit that has been attached to the packet. Thus, we have two separate acknowledgements: `ACK0` and `ACK1`. The first acknowledgement is implemented by sending a real value of 2, and the second acknowledgement is implemented by sending a real value of 1.

3.1.2 Modeling the communication channels

The infrared communication between RCX bricks is facilitated by one byte code instruction that broadcasts a number to all RCXs in the area, excluding itself! This number is stored on each RCX and can be used for computation. For each RCX, we used a variable to model the content of this *infrared buffer*. In our case we have two RCXs: one contains the sender of the protocol and the other contains the receiver. The variable `irs_buf` models the infrared buffer of the sender, and `irr_buf` models the infrared buffer of the receiver.

As the send instruction does not update the local infrared buffer, we use a separate variable for each RCX to model the value that is passed to this instruction. The variable `irs_val` models this value for the sender and the variable `irr_val` models this value for the receiver. Initially, all four variables, `irs_buf`, `irr_buf`, `irs_val` and `irr_val`, are zero.

The sender and receiver now can access the content of the infrared buffer and read the last received value. We model the sending of a value by assigning that value to `irs_val` or `irr_val`. Of course, this sending possibly updates the infrared buffer of the other RCXs in the system. To model the two lossy channels (sender to receiver and vice versa), we used two *channel* processes, see figure 3.1. Let us consider the left channel process that mod-

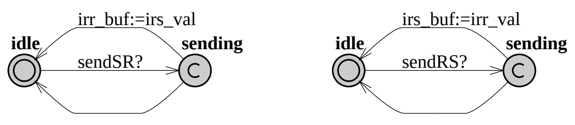


Figure 3.1: `channelSR` on the left and `channelRS` on the right.

els the channel from the sender to the receiver. The channel is activated by synchronization over the urgent channel `sendSR`. Then, the channel process arrives in a committed location in which it can choose to loose the value, or to deliver the value stored in `irs_val` to the other RCX by the assignment `irr_buf:=irs_val`.

Concluding, we use synchronization over the `sendSR` channel in combination with an assignment to `irs_val` to model the broadcast of the value assigned to `irs_val`. For example, the effect of an edge labeled with `sendSR!` and with `irs_val:=5` is that (i) the value of `irs_val` is updated to 5, and (ii) the value of `irr_buf` is maybe updated to 5. Note that this scheme indeed ensures that the infrared buffer of the sender is not updated to 5. Also note that the commitment of the second location of the channel process renders the sending atomic. A similar mechanism is used to model the broadcast of the receiver.

3.1.3 The sender

Figure 3.2 depicts the sender of the protocol. It has been designed to send a packet whose value is stored in the variable `in` to the receiver. It has an

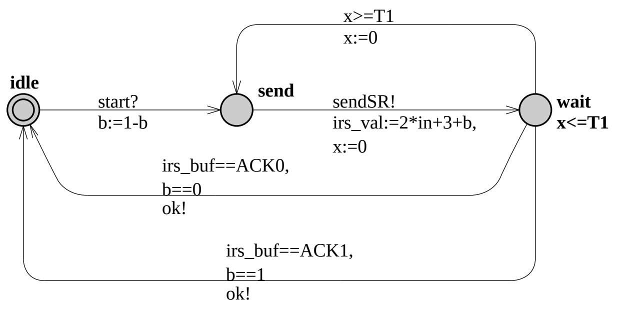


Figure 3.2: The sender: process `S`.

internal clock `x` and one internal variable `b`, the alternating bit of the next packet, that is initialized to 0.

The sender can be started by synchronization over the channel `start` from its initial location `idle`, which flips the alternating bit. In location `send` the sender can synchronize over the urgent channel `sendSR`. Due tot the urgency and the fact that the channel processes are input enabled, this take place without delay. With this edge, the packet is encoded and broadcasted and clock `x` is reset.

In location `wait` the sender waits for a response of the receiver or for a timeout. If an acknowledgment is received that matches the current value of the alternating bit, then the packet has been received. Therefore, the next packet can be send and control is transferred to `idle` by one of the two lower edges.

If a timeout occurs in location `wait`, then the sender assumes that the packet has been lost and retransmits it by entering location `send` again. The timeout constant is fixed by the constant `T1`, which equals 1000000 time units. This large value originates from our time scale: we use one million UPPAAL time units to model one second. Thus, our timeout constant equals 1 second. Note that the sender cannot guarantee the arrival of a packet, since the channel may loose all packets.

3.1.4 The receiver

Figure 3.3 depicts the receiver. It has one internal variable `b`, the expected value of the alternating bit of the next packet, that is initialized to 1. In

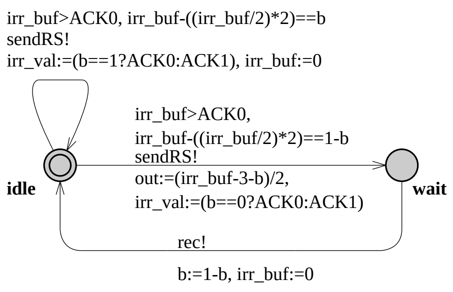

irr_buf>ACK0, irr_buf-((irr_buf/2)*2)==b
sendRS!
irr_val:=(b==1?ACK0:ACK1), irr_buf:=0
irr_buf>ACK0,
irr_buf-((irr_buf/2)*2)==1-b
sendRS!
out:=(irr_buf-3-b)/2,
irr_val:=(b==0?ACK0:ACK1)
idle
wait
rec!
b:=1-b, irr_buf:=0

Figure 3.3: The receiver: process `R`.

the location `idle` it waits until the value of its infrared buffer, modeled by `irr_val`, becomes larger than `ACK0`. Our encoding scheme of packets and acknowledgements guarantees that a packet has been transmitted by the sender. However, if the alternating bit of this packet does not match the expected bit, an acknowledgement with a flipped bit is send back. This has been modeled by the edge from `idle` to itself.

If the alternating bit of the packet matches the expected bit, then the packet is decoded and assigned to variable `out`. Moreover, an acknowledgement is send back and control is transferred to location `wait`.

In location `wait` the receiver just waits for another process to use the received packet. This is necessary to prevent packet loss: the variable `out` may not be overwritten until the packet in it has been processed. In other words, the receiver has a buffer with size one.

Whenever the receiver synchronizes over channel `rec` it assumes that the packet has been processed. Now it toggles the value of the expected bit and transfers control to location `idle`: it is ready to receive another packet and to place it in `out`.

In the literature, the receiver sends an acknowledgement *after* the processing of the packet (thus after the `rec!` synchronization). This prevents the behavior that the sender must resend a packet despite correct delivery of all packets/acknowledgements, which is present in our model. Our choice originates from a specific property of the translation – which we explain in chapter 4 – namely that every location introduces extra overhead in the generated byte code. Thus, if we follow the more logical scheme of the literature, then we would need three locations instead of two, with the result of additional overhead in the generated code.

3.1.5 The tester

Figure 3.4 depicts a test process for the protocol. It uses two arrays, `input` and `output`, both with length `N`. Moreover, it has two internal array indices: `i` is used for the input array and `j` is used for the output array. The purpose

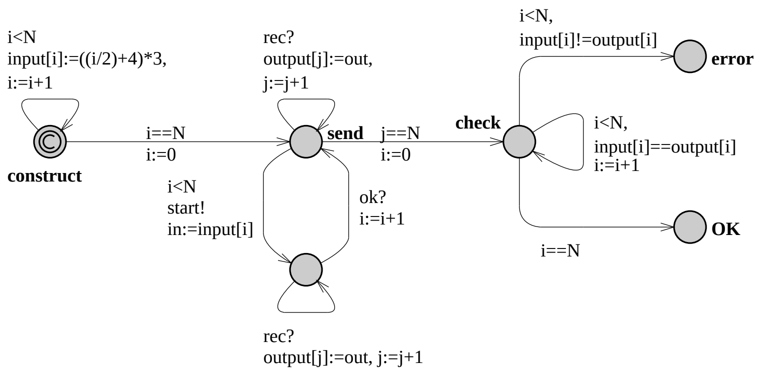


Figure 3.4: A tester for the protocol: process `Tester`.

of the tester is to send several packets and check (i) whether or not the input is equal to the output, and (ii) whether or not the order is preserved.

In location `construct` the tester fills the input array with some predetermined values greater than zero, and transfers control to location `send`. From this location, the tester starts the protocol `N` times, using the values of the input array as input for the sender. Each time the receiver has successfully received a packet, it can synchronize over the `rec` channel, filling the next entry in the output array of the tester.

Finally, if all packets have successfully been received, control is transferred to location `check`. This location compares the input and output arrays. If they differ, then the location `error` is reachable.

Note that our tester does not test all possible ways to use the protocol due to two issues. First, the number of consecutive runs of the protocol is fixed by the constant `N`. Second, the values of the input array are also fixed. To gain trust in the protocol, we set `N` to 5 and as a consequence the values in the input array are 12, 12, 15, 15 and 18. This input array tests two equal packets after each other and two different packets after each other.

3.1.6 Validation of the protocol

The protocol must provide a reliable and in-order data-stream over an unreliable channel. As mentioned in the previous section, we can only validate this for the tester, and not in general. However, we are confident that our tester gives a good representation of the context in which we will use the protocol.

As a first requirement, we want that the protocol actually delivers packets: the input array of the tester can be transferred to the output array using the protocol. This is expressed by property (3.1).

$$\exists\Diamond(\texttt{Tester.OK}) \tag{3.1}$$

This is not enough, however, since executions may exist such that the input array does not match the output array. We require that this does not happen: the location `error` of the tester must be unreachable, which is expressed by property (3.2).

$$\forall\Box(\neg\texttt{Tester.error}) \tag{3.2}$$

Fortunately, both properties are satisfied by the model; model checking is a matter of tenths of a second.

It is not very difficult to use the design of the protocol as a part of other designs, since the sender and the receiver have clearly defined interfaces. The sender uses the channels `start` and `ok` and the variable `in`. The receiver uses the channel `rec` and the variable `out`. In sections 3.2 and 3.3 we demonstrate how we can easily use this protocol in other designs.

3.2 The Level Crossing

The level crossing consists of a railroad track with a train and a gate, both controlled by RCX bricks. The main idea is that if the train is at the gate, then the gate must have been lowered. Moreover, we want the gate sometimes to be raised to let other traffic cross the railroad.

The RCX of the gate can lower and raise the gate and it can control warning lights. It is clear that the gate controller must react to the position of the train in order to achieve the sketched goals. The gate controller uses three sensors to that end. First, a touch sensor that senses if the gate has been completely lowered or not. Second, a light sensor positioned before the gate along the railroad track that senses the approach of a train. Third, a light sensor positioned after the gate along the railroad track that senses the departure of a train.

The RCX of the train uses only one actuator: the engine of the train. It can be switched on and off to start or stop the movement of the train.

The difficulty in this scenario is that the gates may not lower correctly. If that happens, then the gate controller must send a message to the train such that it can take the appropriate action of halting. The communication between the gate and the train is facilitated by the communication protocol explained in the previous section.

In the next section we first introduce a very simple process, called the hurry dummy, that enables us to make edges “urgent”. Next, we describe the models of the physical gate and the physical train. Finally, we describe the models of the controllers for these physical processes, and we state the validation properties.

3.2.1 The physical gate and the physical train

Our design contains processes that model the physical gate and the physical train. These processes provide the input for the three sensors of the gate. As in the conceptual design of the communication protocol we use one million Uppaal time units to model one second.

The physical gate, depicted in figure 3.5, models the behavior of the gate. It has an internal clock `x`. Its behavior depends on whether the engine of the gate is on or off and on the direction of the engine. The operational mode of the engine is modeled by the variable `gm`, that either has the value `ON` or `OFF` with the obvious meaning. The direction of the engine is modeled by the variable `gd`, that either has the value `REV` or `FWD`.

All edges to the distinct error location are reactions to unwanted situations. For example, the edge from the initial location `up` to the location `error` is taken if the direction of the engine is such that the gate will be raised even further, which is physically impossible, and the engine is turned on.

In the location `lowering`, the physical gate can choose to lower with success, or to fail. If it failed – control is in the location `failure` – then the physical gate can at any moment choose to repair the gate and transfer control to location `down_off`. In this location the gate has been lowered and its engine is switched off. If the engine has the appropriate direction and its is switched on again, the gate is raised, and control can eventually return to the initial location `up`.

Note the updates of the sensor value of the touch sensor, modeled by assignments to `sns1`. The edges from `lowering` to `down_on` and the edge from `failure` to `down_off` both increase the pressure on the sensor, since the physical gate is pushing on it. If the gate is raised again, then the edge

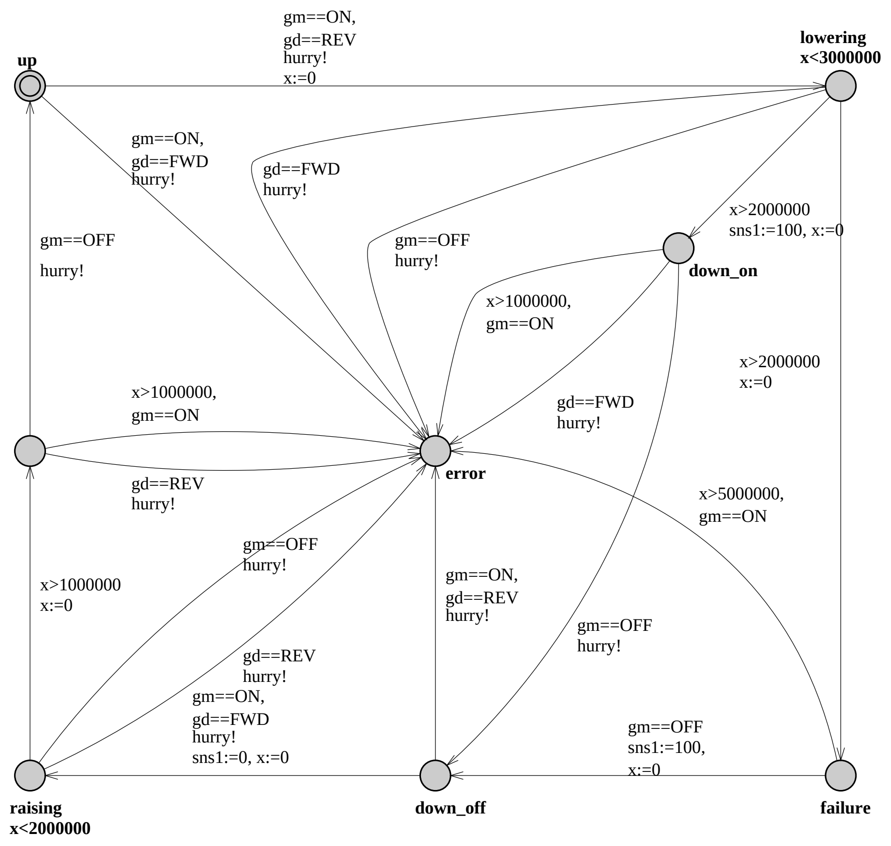


Figure 3.5: The physical gate.

from `down_off` to `raising` is taken and the pressure is relieved.

The physical train, depicted in figure 3.6, models the position of the train using an internal clock `x`. Its behavior depends on whether or not the the engine of the train is on or off. The operational mode of the engine is modeled by the variable `engine`.

The basic idea is that the physical train just drives in circles and halts whenever the engine is turned off. If the engine is turned on again, the train resumes its journey. The physical train updates the values of the light sensors, modeled by `sns2` and `sns3`, when it passes them.

Note that this model of the physical train is not very accurate. This is due to the fact that whenever the train halts, the position – modeled by the clock `x` – *cannot* be remembered, since clocks cannot be stopped and their value cannot be assigned to an integer variable. Of course, we could create multiple "halting" locations for each "driving" location to remember the interval in which `x` was at the halting time. However, this can also never be exact, and it creates a large(r) model. Hybrid or stopwatch automata would be very suitable to model the movement of the train.

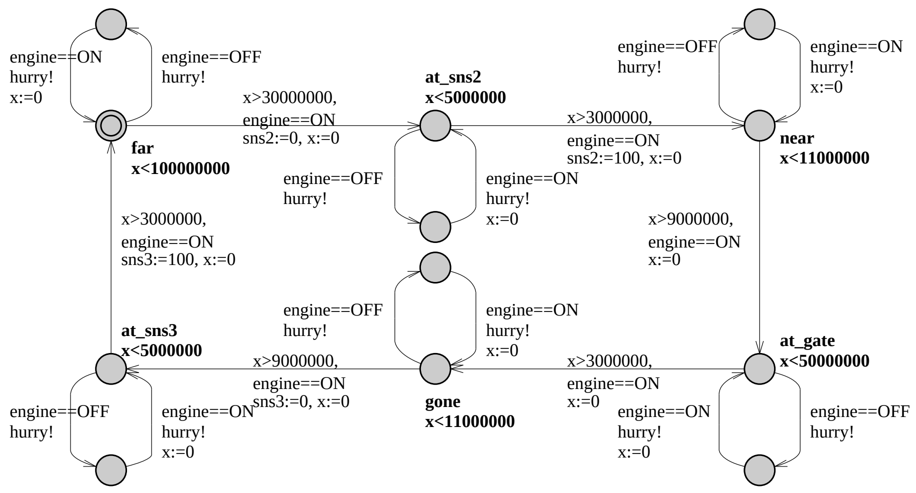


Figure 3.6: The physical train.

The constants that appear in clock guards and invariants of the models of the physical train and gate result from experiments with the actual LEGO systems.

3.2.2 The gate controller

The gate controller actually consists of two processes: the sender of the communication protocol, depicted in figure 3.2 on page 19, and a main controller, depicted in figure 3.7. This main controller has one internal clock `t`. It uses the two light sensors, whose values are modeled by `sns2`

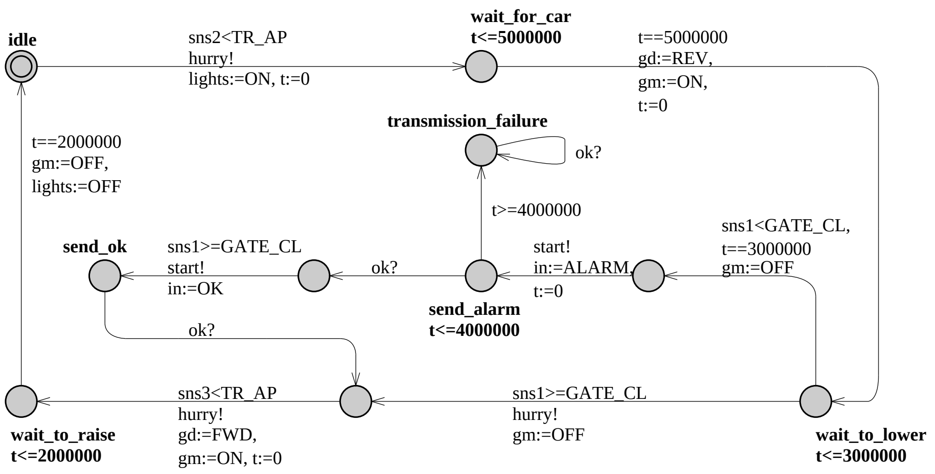


Figure 3.7: The main controller of the gate.

and `sns3`, and the touch sensor, modeled by `sns1`. The actuators used by the gate are warning lights, whose operational mode is modeled by `lights`, and the gate engine, whose operational mode is modeled by `gm` and whose direction is modeled by `gd`.

The constant `TR_AP` is a threshold value for the light sensors. If the value of a light sensor is smaller that this threshold, then the train is at that light sensor. Similarly, the constant `GATE_CL` is a threshold value for the touch sensor. If its value is larger than this threshold, then the gate has been lowered successfully. Again, these thresholds have been determined by experiment.

Normally, the operation of the gate starts in the location `idle` and is as follows. Whenever it senses the approach of a train (the guard `sns2<TR_AP` is satisfied), the gate turns on its warning lights and waits 5 seconds to let traffic pass that is on the track (location `wait_for_car`). After these 5 seconds, it starts lowering the gate (location `wait_to_lower`). If the gate senses the successful lowering of the gate (the guard `sns1>=GATE_CL` is satisfied), it takes the lower edge and waits for the departure of the train. After this signal has been received (the guard `sns3<TR_AP` is satisfied), the gate is raised (location `wait_to_raise).` Finally, the gate turns off its warning lights and returns to the initial location.

The previous scenario assumed that the lowering of the gate is successful. However, the engine of the lowering and raising mechanism might fail. To detect this, the gate uses a timeout. If the gate is in the location `wait_to_lower` and the signal from the touch sensor is not received within 3 seconds, then the gate assumes failure of the engine and starts an emergency scenario.

The emergency scenario consists of three steps. First the gate sends, using the sender of the communication protocol, an alarm message to the train to inform the train about the situation. If the train receives this alarm message, it should halt to guarantee that it will not arrive at the gate when the gate has not been lowered. However, the communication is unreliable and the communication protocol that we discussed earlier cannot guarantee that a message eventually arrives. Thus, if the message is not successfully delivered within four seconds, then a transmission failure occurred and the gate enters an emergency location (`transmission_failure`) that cannot be left.

If the alarm message has been delivered, then the gate knows that the train has been halted and it waits for the repair of the engine (which is modeled by the process of the physical gate). If the engine has been repaired and the gate has been lowered, then it sends an ok message to the train to signal that it can start moving again. The main controller waits until it knows that the train has received the message, after which the operation of the gate controller continues as normal.

3.2.3 The train controller

The train controller consists of two processes: the receiver of the communication protocol, depicted in figure 3.3 on page 20, and a controller, depicted in figure 3.8. In the previous section we mentioned that the train should

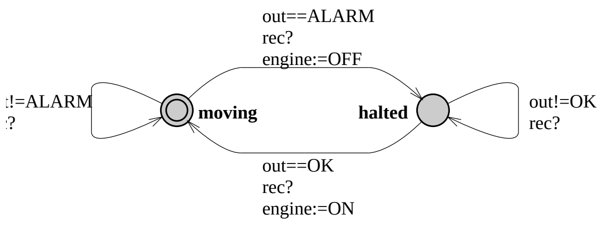


Figure 3.8: The main controller of the train.

halt if it receives an alarm message and that it may resume its journey if it receives an ok message. It is easy to map this behavior to the process in figure 3.8. Note that the train controller is input enabled with respect to the synchronizations of the receiver of the communication protocol.

As mentioned in this and the previous section, we use the communication protocol described in section 3.1. Therefore, the channel processes are also present in this system. The tester, however, is not present since the main controllers of the train and the gate interface with the receiver respectively sender.

3.2.4 The hurry dummy: urgent edges

The hurry dummy process has been depicted in figure 3.9. The only use of this process is to create *urgent edges*. Since the channel `hurry` is an urgent

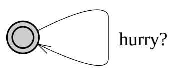


Figure 3.9: The hurry dummy.

channel and synchronization over it is always enabled, any edge that has been labeled with `hurry!` is taken as soon as possible: no time may elapse if such an edge is enabled.

3.2.5 Validation of the Level Crossing

We verified some properties of our conceptual design of the Level Crossing. Property (3.3) ensures that if a deadlock occurs, then the main controller of the gate is in the location transmission_failure.

$$\forall\Box(\texttt{deadlock} \Rightarrow \texttt{GateController.transmission\_failure}) \tag{3.3}$$

Thus, if the main controller of the gate is not in this location, then no deadlock occurs.

Property (3.4) ensures that whenever the train arrives at the gate, the gate has been lowered or a transmission failure has occurred.

$$\begin{aligned}\forall\Box(&\texttt{PhysicalTrain.at\_gate} \Rightarrow \\ &(\texttt{PhysicalGate.down\_off} \vee \\ &\texttt{GateController.transmission\_failure}))\end{aligned} \tag{3.4}$$

It is noteworthy that this property is not satisfied if the main controller of the gate is allowed to wait for six seconds in location send_alarm.

However, this property is meaningless if the location at_gate of the physical train is not reachable. Therefore, we need property (3.5) that states that this location *is* reachable.

$$\exists\Diamond(\texttt{PhysicalTrain.at\_gate}) \tag{3.5}$$

The last property that we verified, property (3.6), ensures that the physical gate is treated well by the main controller of the gate.

$$\forall\Box(\neg\texttt{PhysicalGate.error}) \tag{3.6}$$

Fortunately, all four properties are satisfied by our conceptual design of the level crossing. The model checking did not take long on a standard desktop computer: properties (3.3), (3.4) and (3.6) can be checked within a few seconds with the convex hull approximation. Property (3.5) can be checked within one second without the convex hull approximation.

Note that we cannot ensure that the gate is raised after it has been lowered. This is due to the fact that there is no way of ensuring that the messages of the gate to the train eventually arrive at the train. In section 3.2.2 we mentioned that – after failure and repair of the gate – the gate controller tries to send the ok message to the train controller until it is received correctly. It is possible that all tries of the sender fail due to loss of messages.

3.3 The Production Cell

The production cell is slightly more complicated than the level crossing of the previous section. It is schematically depicted in figure 3.10. The purpose of the production cell is to transport all bricks, that arrive by belt B, to the container using a rotating robot arm. The robot arm can rotate, from its depicted position, 180 degrees in a counter clockwise direction. The robot arm can pickup a brick by activating a magnet that is positioned above the brick. Magnet 1 is used to transport bricks from the pickup point above

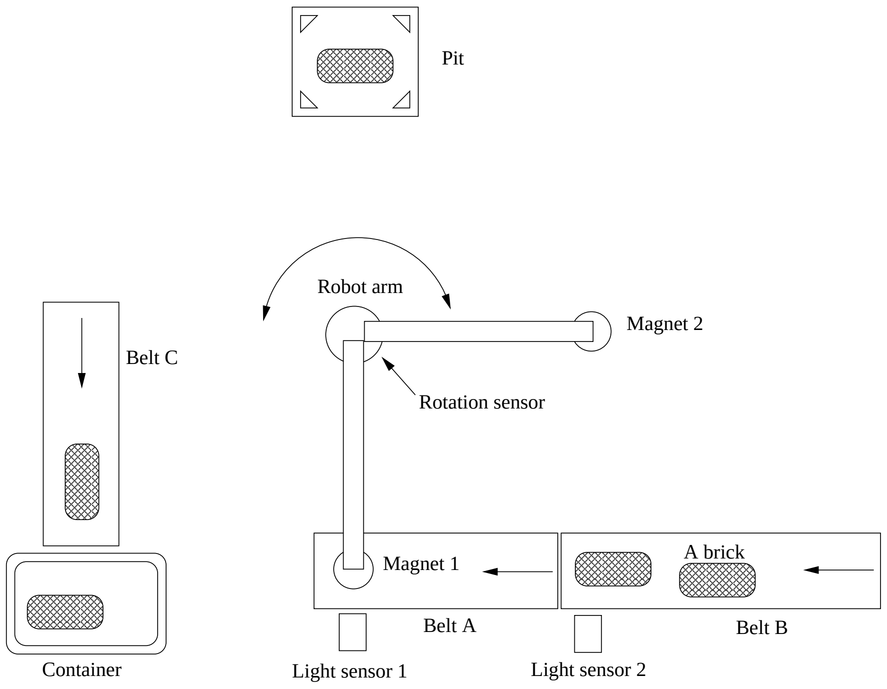


Figure 3.10: The production cell.

belt A to the pit. Magnet 2 is used to transport bricks from the pit to the delivery point above belt C.

The system is controlled by two RCX bricks. The first RCX controls the robot arm. It uses a rotation sensor to determine the position of the robot arm. It uses three actuators: one engine to rotate the robot arm and two magnets. The second RCX controls the three belts in the system and it uses the two light sensors to detect bricks.

3.3.1 The physical robot arm

Figure 3.11 depicts our model of the physical robot arm. It has an internal clock `x` that is used to model the rotation by updating the value of the rotation sensor, modeled by the variable `rotation_sensor`, of the robot arm in discrete steps. The relative speed of the robot arm is fixed by the constant `SPEED`, which has a value of 333333 time units. Again, this large constant is due to our time scale: one second equals one million time units. The behavior of the process depends on the operational mode of the engine of the robot arm, which is modeled by the variable `arm`, and on the direction of this engine, which is modeled by the variable `arm_dir`.

Note that many edges have been labeled with `hurry!` to ensure immediate reaction to change of, e.g., engine modes and directions. Thus, the hurry dummy process is also a part of the system, but we do not depict it again.

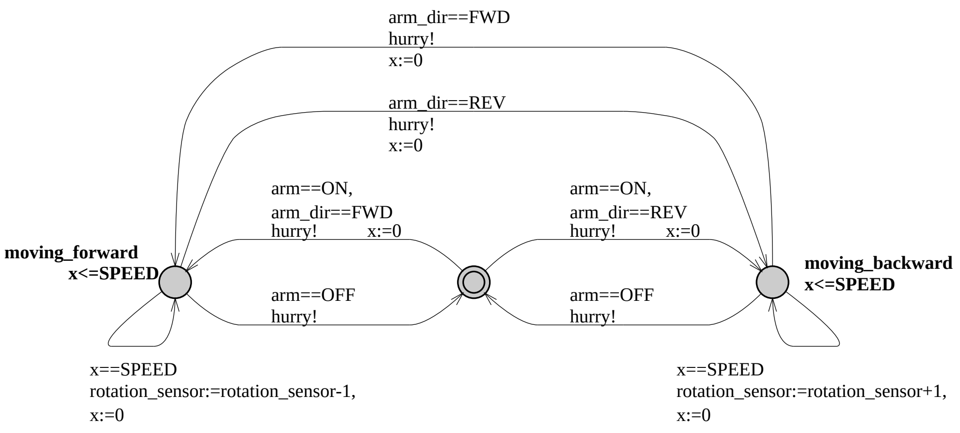


Figure 3.11: The physical robot arm.

3.3.2 The brick template

Figure 3.12 depicts the template which models how a brick passes through the system. For validation purposes, an arbitrary number of bricks may be present in the system. However, the laws of physics prevent these bricks from taking up the same space. As a result, they cannot arrive at light sensor 2 at the same moment. To avoid this behavior, the bricks all share and use the semaphore `s` for arriving at this light sensor.

Note that the belt controller ensures that there is at most one brick on belt A. The robot arm controller ensures that there is at most one brick on a magnet and in the pit. Therefore, belt B *was* the only place where our model allows bricks to take up the same space, and we do not need to apply the semaphore trick anywhere else.

The behavior of a brick depends on many variables. First there are `beltA` and `beltB` that model the operational modes of the engines that move belt A and belt B. Then there are `mag1` and `mag2` that model the operational modes of the magnets on the robot arm. For example, consider the location `at_sensor_1_safe`. The brick can move to location `on_mag1`, if the robot arm is placed above the end of belt A (modeled by the guards concerning the variable `rotation_sensor`), and the magnet must of course be switched on.

The constant `deliver`, `pit` and `pickup` are used in combination with the rotation sensor to determine the position of the physical robot arm. Again, the values have been experimentally determined.

We noted that the belt controller sees bricks using the light sensors. Therefore, bricks "update" the values of these light sensors, that are modeled by the variables `light_sensor_1` and `light_sensor_2`. For example, if a brick enters the location `at_sensor_2`, then it updates the value of the corresponding sensor. When it has completely passed light sensor 2, it updates the sensor value again. This happens on the edge from location

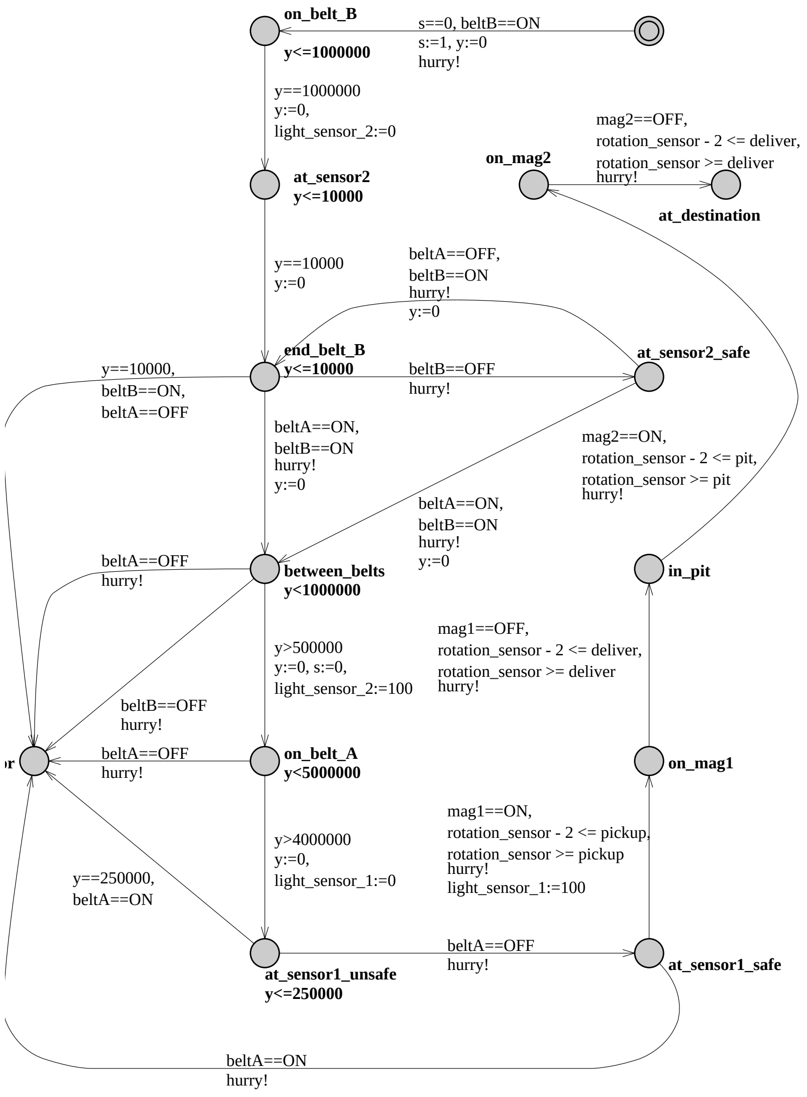


Figure 3.12: A brick.

`between_belts` to `on_belt_A`.

Again, there exists a distinct error location that is reachable if bricks are maltreated. For example, if a brick is between belt A and belt B (location `between_belts`), then both belts must keep moving.

3.3.3 The robot arm controller

The robot arm controller consists of three processes: the receiver of the communication protocol, depicted in figure 3.3 on page 20, a message handler, and the main controller. We start with the message handler, depicted in figure 3.13. Whenever the receiver synchronizes over the channels `rec`, the message handler updates the variable `brick_ready` that signals the presence

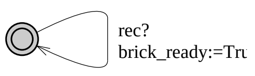


Figure 3.13: The message handler of the robot arm controller.

of a brick at the end of belt A. This synchronization is not guarded, since the sender of the protocol transmits only one “type” of packet.

Figure 3.14 depicts the main controller. Its only function is to pick bricks up at belt A and deliver them to belt C. It uses three internal variables, `brick_on_mag1`, `brick_on_mag2` and `bricks_in_pit`, all with obvious meaning. The first two variables are initialized to `False`, whereas the last variable is initialized to zero. We sketch the operation of the main con-

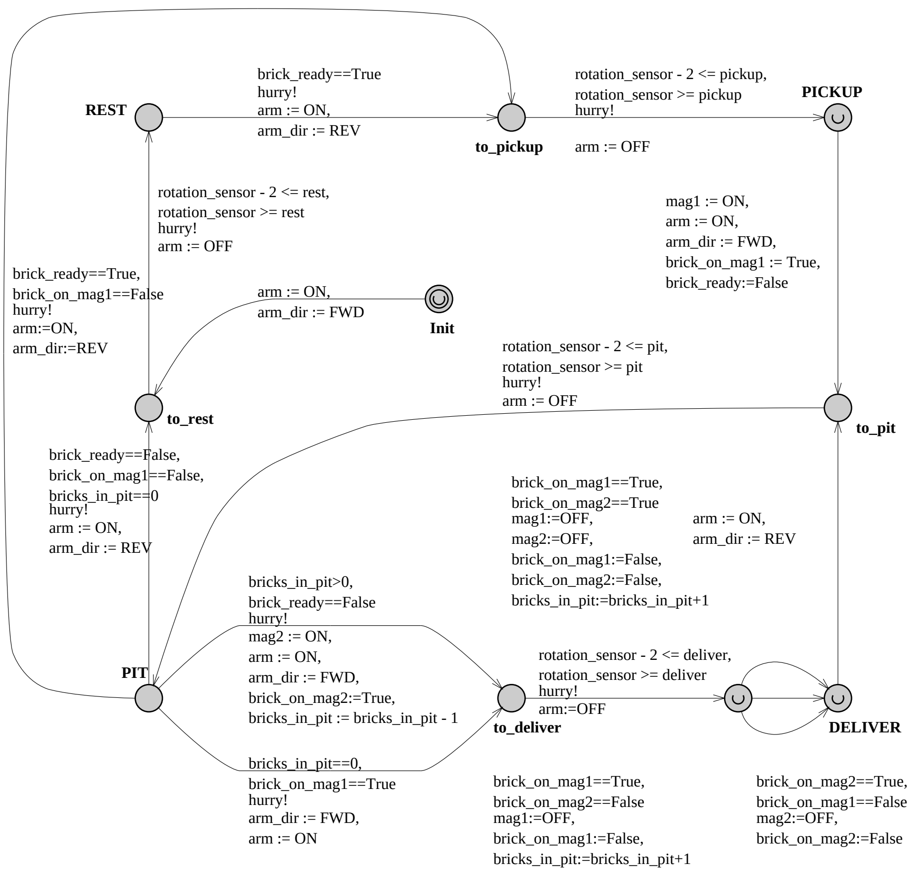


Figure 3.14: The main controller of the robot arm.

troller by paths through the process. We start with an explanation of the position of the robot arm with respect to the locations in the process:

- `REST`: When in this location, the position of the robot arm is obtained by a counter clockwise rotation of about 45 degrees of the position of the robot arm in figure 3.10.

- `PICKUP`: The arm with magnet 1 is above belt A, as depicted in figure 3.10.

- `PIT`: The arm with magnet 2 is above the pit.

- `DELIVER`: The arm with magnet 1 is above the pit and the arm with magnet 2 is above belt C.

Before the system can be started, the robot arm must be placed in the position as in figure 3.10. The transition from `Init` to `to_rest` is taken immediately due to the urgency of `Init` and the engine is started. The robot arm rotates about 45 degrees in the counter clockwise direction. As soon as this resting position is reached, the transition to `REST` is taken and the engine is turned off.

In the location `REST`, the robot arm controller waits for the `BRICK_READY` message from the belt controller, which means that a brick is positioned at the end of belt A. As soon as this message is received, the robot arm controller moves to the pickup point and enters location `PICKUP`. In this location, magnet 1 is hovering above the brick and when it is switched on the brick will stick to it. Next, the robot arm must move the brick on magnet 1 to the pit and it switches on its engine and enters location `to_pit`.

When the value of the rotation sensor is such that the robot arm controller may conclude that magnet 2 is above the pit, the engine of the robot arm is switched off and the location `PIT` is entered. Now there are four possibilities:

- If there is no brick on magnet 1 and there is no brick in the pit, then the robot arm controller returns to its resting position.

- If a brick is ready for pickup and there is no brick on magnet 1, then the robot arm starts moving to the pickup position.

- If no brick is ready and there is a brick in the pit, then this brick is picked up by magnet 2 and the robot arm starts moving to the delivery position.

- If there is no brick in the pit, but there is a brick on magnet 1, then the robot arm also starts moving to the delivery position.

When the robot arm arrives at the delivery position, there are three possibilities: there is a brick on magnet 1, there is a brick on magnet 2, or there are bricks on both magnets. In all cases, the bricks are dropped by switching off the magnets and the robot arm starts moving back to the pit position.

3.3.4 The belt controller

The belt controller consists of two processes: the sender of the communication protocol, depicted in figure 3.2 on page 19, and a main controller, depicted in figure 3.15. For a correct functioning of the system, the belt controller must ensure that at any time at most one brick is on belt A. Moreover, when a brick arrives at light sensor 1, belt A must be stopped and the robot arm controller must be informed that a brick is ready for pickup. The belt controller has an internal clock `t` and an internal boolean

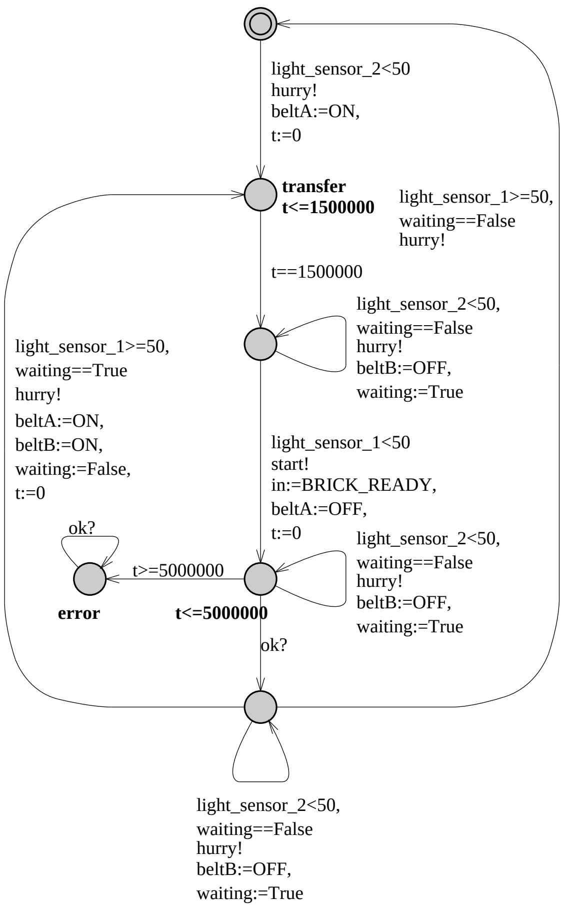


Figure 3.15: The main controller of the belts.

variable `waiting` that is initialized to `False`. This boolean indicates whether or not a brick is waiting at light sensor 2.

In the initial location, belt A has been stopped and belt B and C are running. When the controller sees a brick pass light sensor 2, belt A is started and the brick is transfered from belt B to belt A. After this transfer, belt B must be stopped if another brick arrives at light sensor 2. If the first brick arrives at light sensor 1, then belt A is stopped and synchronization over the channel `start` starts the sender of the communication protocol to inform the robot arm controller of the presence of the brick. Clock `t` is reset with this transition. If the transmission is not successful within five seconds, then the belt controller enters the location `error`. Otherwise, it resumes with waiting for the robot arm to pickup the brick.

3.3.5 Validation of the Production Cell

We used two instances of the brick template in the validation. Property (3.7) assures that the bricks always arrive at their destinations or that a transmission failure put the belt controller in the error location.

$$\begin{aligned}&\forall\Diamond((\texttt{Brick1.at\_destination} \wedge \texttt{Brick2.at\_destination})\vee\\&\quad \texttt{BeltController.error})\end{aligned} \tag{3.7}$$

Note that this is a very expensive property to check, which we only added to show the expressiveness of the $\forall\Diamond$ path property.

Property (3.7), however, does not guarantee that the bricks arrive at their destination, since it could be the case that there is a transmission error for all paths. To ensure that this is not true, we also state property (3.8) that means that at least one trace exists that leads to the situation where both bricks are at their destinations.

$$\exists\Diamond(\texttt{Brick1.at\_destination} \wedge \texttt{Brick2.at\_destination}) \tag{3.8}$$

It is interesting to derive an upper bound on the time it takes to transport the bricks to their destinations. Property (3.9) specifies this upper bound as 48 seconds. This property is not satisfied for a value of 47 seconds.

$$\begin{aligned}&\forall\Box(\texttt{rt>=48000000} \Rightarrow\\&\quad (\texttt{Brick1.at\_destination} \vee \texttt{BeltController.error}))\end{aligned} \tag{3.9}$$

Note that there is no need to include the second brick in this property, since the bricks behave exactly the same. So if this property is satisfied for the first brick, then it is also satisfied for the second brick.

Property (3.10) ensures that the bricks are not maltreated. This property also holds for all other bricks in the system by symmetry.

$$\forall\Box(\neg\texttt{Brick1.error}) \tag{3.10}$$

The magnets on the robot arm can only carry one brick at a time. The properties in equation (3.11) assure that this also is the case for our model.

$$\begin{aligned}&\forall\Box(\neg(\texttt{Brick1.on\_mag1} \wedge \texttt{Brick2.on\_mag1}))\\&\forall\Box(\neg(\texttt{Brick1.on\_mag2} \wedge \texttt{Brick2.on\_mag2}))\end{aligned} \tag{3.11}$$

The pit also has a capacity of one brick. The robot arm controller has an local variable that counts the number of bricks in the pit. Property (3.12) assures that this number does not exceed one.

$$\forall\Box(\texttt{RobotarmController.bricks\_in\_pit} \leq 1) \tag{3.12}$$

Property (3.13) guarantees that our production cell keeps working until either the belt controller is unable to transmit a message to the robot arm controller within 5 seconds, or all bricks are at their destinations and the robot arm is in the resting position.

$$\begin{aligned}&\forall\Box(\texttt{deadlock} \Rightarrow (\texttt{BeltController.error} \vee \\ &\quad(\texttt{RobotarmController.REST} \wedge \\ &\quad\texttt{Brick1.at\_destination} \wedge \texttt{Brick2.at\_destination})))\end{aligned} \tag{3.13}$$

Finally, we mentioned that there must never be two bricks on belt A at the same time, since this would lead to collisions due to the rotational arc of the robot arm. Property (3.14) assures this mutual exclusion on belt A.

$$\forall\Box(\neg(\texttt{Brick1.on\_beltA} \wedge \texttt{Brick2.on\_beltA})) \tag{3.14}$$

All properties are satisfied by our design and we conclude that the system functions as required. The model checking did not take long on a standard desktop computer. Properties (3.10)-(3.14) can be checked within one minute with the convex hull approximation. For some unknown reason, model-checking of property (3.9) takes more time: about five minutes. Model-cheking property (3.8) takes less than a minute with a depth-first search order and without the convex hull approximation. The only remaining property, property (3.7), is very expensive: We terminated the model checking after 18 hours of computation time and a memory usage of 2289 MB.

3.4 Summary

This chapter introduced two case-studies that are used to evaluate our technique for obtaining correct reactive programs for the RCX using the model checker Uppaal.

We started with the conceptual design of a communication protocol that is very similar to the alternating bit protocol. This "stop-and-wait" protocol provides a reliable data stream over a lossy channel. Therefore, it is very well suited for the communication between different RCX bricks, since the byte code language – the language used to program the RCX – only offers an unreliable send primitive that cannot tell whether or not the message has been received by other RCXs.

The two real case studies, the *Level Crossing* and the *Production Cell*, each consist of a LEGO setup with two RCXs. In both systems the RCXs must communicate with each other and that is where our communication protocol appeared again. We globally described the Uppaal models that constitute the conceptual designs of the systems and we stated our requirements on the systems in terms of the Uppaal specification language. Fortunately, all our requirements on the conceptual designs are satisfied and the model checking of the invariance and reachability properties took little time on a standard desktop computer.

The conceptual design of the systems took experiments with the actual LEGO setups to measure timing parameters for the processes of the physical world. For example, the value of `SPEED`, used in the process of the physical robot arm, originates from measurement of the real rotation speed of the arm. As for the construction of the control processes, the simulator in UPPAAL proved to be very helpful. Also the option to generate diagnostic traces that disprove properties is useful: it enabled us to quickly debug the communication protocol.

In the next chapter we describe our translation from the conceptual designs in the UPPAAL modeling language to executable byte code for the RCX and to another UPPAAL model of the run-time behavior of the generated byte code. This second model can be used to check whether or not the *implementation* of our conceptual design functions as we expect.

Chapter 4

Compiling conceptual designs

This chapter describes the heart of our technique for obtaining correct reactive programs for the RCX using the model checker UPPAAL. We define a translation from UPPAAL models to RCX byte code and an UPPAAL model of the run-time behavior of that byte code.

In section 4.1 we discuss why additional information must be present in an UPPAAL model to support the implementation on the RCX. Moreover, we define what this additional information should be. In section 4.2 we first discuss how we generate executable byte code and the model of that byte code. Next, we link the input and the output of the translation and we discuss the restrictions that result from our choices. In section 4.3 we discuss our implementation of the translation – the compiler `uppaal2rcx` – and give a small illustrative example. Finally, in section 4.4 we apply our compiler to the Level Crossing and the Production Cell. We test the generated byte code on the physical LEGO setups and we *try* to model check it.

4.1 Extra information in conceptual designs

In this section we discuss the extra information that must be present in UPPAAL models to support their implementation. This need originates from the fact that modeling tools in general, and UPPAAL in particular, are not specifically designed to model reactive programs for some platform. Therefore, the models that can be constructed with these tools must explicitly be mapped to the target platform and this is, in general, not possible without additional information.

4.1.1 Only a UPPAAL model is not enough

A UPPAAL model that is a conceptual design of a system for the RCX contains aspects that model specific properties of the RCX, e.g., a sensor value of the RCX might be modeled by an integer variable in the UPPAAL model. Moreover, the UPPAAL model might contain control programs for more than one RCX brick. This is clearly illustrated by the case study of the Production Cell. Both RCX bricks should be modeled, since the correct functioning of the system depends heavily on the interaction between the RCX bricks.

A last point concerns the fact that the Uppaal model contains processes that should not be translated. In the Production Cell for example, the channel processes model the behavior of the lossy communication between RCX bricks and the hurry dummy is a helper process, which provides an urgent synchronization channel that is always available. It is clear that these processes should not be implemented. The following problems thus occur when we want to translate some Uppaal model to RCX byte code [Hen01]:

- Distribution. We must be able to tell which processes should be implemented, and on which RCX brick they should be implemented.
- We must be able to tell how the implementation details of the RCX are modeled in the Uppaal model. For example, a user may model a sensor type by a Uppaal variable. To be able to translate these additional "semantics" of the variable, the user must explicitly tell that the variable models a sensor type.

Note that these problems also occur when other target platforms or model checkers are used. To solve these problems, a model should contain more information, namely so-called *type mappings*. The next subsection explains this more carefully.

4.1.2 The type mappings

The RCX can use four timers, 32 integer variables, one infrared communication channel, three sensors and three actuators. It is a natural choice to model the timers of the RCX with Uppaal clocks, and the RCX integer variables with the Uppaal bounded integer variables.

Each sensor is defined by a tuple (t, m, v), where t is the sensor type, m is the sensor mode and v is the value of the sensor, all integers. Similarly, each actuator is defined by a tuple (m, d, p), where m is the operational mode, d is the direction and p is the power of the actuator. Therefore, it is natural to model these sensor and actuator attributes with Uppaals bounded integer variables.

Modeling the infrared communication channel is a bit complicated due to the possibilities of the byte code language. As already mentioned in section 3.1.2, the byte code language only supports a broadcast instruction that *excludes* the sending RCX. It also contains an instruction to update the infrared buffer of the RCX. It may seem that a "total" broadcast can be achieved by the combination of these two instructions. However, the argument for the first instruction may be computed at run-time, whereas the argument for the second instruction must be known at compile-time. Therefore, we model this infrared communication using two integer variables. Broadcasting is modeled by assigning a value (that may be computed at run-time) to the first variable. This variable may not be used in guards, as it is not stored on the RCX. The second variable models the actual local infrared buffer, which can be used in guards and in assignments. However, if it is the target of an assignment, then the assigned value should be constant.

Summarizing, we assume the following concerning the UPPAAL model:

- The timers of the RCX are modeled by clocks.
- The sensor and actuator attributes and regular variables of the RCX are modeled by integer variables.
- The IR communication is modeled by reading from and writing to two integer variables.

We do not think that these assumptions are limiting, which is motivated by the case-studies of the Level Crossing and the Production Cell.

As explained in the previous section, we must map parts of the UPPAAL model to details of the RCX (e.g. sensor values, actuator modes etcetera), which are covered by the set T_{RCX}:

$$
\begin{aligned}
T_{RCX} = \{ \ & \textit{sns\_1\_type, sns\_1\_mode, sns\_1\_val, sns\_2\_type, sns\_2\_mode,}\\
& \textit{sns\_2\_val, sns\_3\_type, sns\_3\_mode, sns\_3\_val, out\_1\_mode,}\\
& \textit{out\_1\_dir, out\_1\_power, out\_2\_mode, out\_2\_dir, out\_2\_power,}\\
& \textit{out\_3\_mode, out\_3\_dir, out\_3\_power, ir\_buffer, ir\_value,}\\
& \textit{regular} \ \}
\end{aligned}
$$

Moreover, the UPPAAL model has to be distributed over the various RCXs, and processes that model the environment should not be implemented. The mapping of RCX details and the distribution of processes are facilitated by three *type mappings*, which we define next.

Let V denote the set of bounded integer variables of a UPPAAL model, and let N denote a set of names. A partial mapping must exist that maps a subset of the variables in the model to an RCX brick and a RCX detail:

$$\mathit{Type}_V : V \hookrightarrow N \times T_{RCX}$$

Variables that are not mapped to a type, are part of the environment. For example, consider the infrared buffer variable `irr_buf` of the Production Cell: $\mathit{Type}_V(\texttt{irr\_buf}) = (\texttt{RobotarmController}, \mathit{ir\_buffer})$.

A similar partial mapping must be present for the clocks, denoted by X, that are used in the model:

$$\mathit{Type}_C : X \hookrightarrow N$$

Again, clocks that are not mapped to a RCX are part of the environment.

To distribute the various processes over the RCXs and the environment, there must be a final mapping that directs each UPPAAL process in the model to some RCX, or to the environment. Let A denote the set of processes in the following definition:

$$\mathit{Type}_A : A \rightarrow \{\mathit{environment}\} \cup \{\, (\mathit{rcx}, n) \mid n \in N \,\}$$

For example, the three processes that constitute the robot arm controller all have the same type, e.g., $(rcx, \texttt{RobotarmController})$. On the other hand, the hurry dummy has type *environment*. From now on, we call a process that has type (rcx, n) a *control program process*. A process with type *environment* is called an *environmental process*.

Note that the inclusion of the name of the target RCX in the type mapping $Type_V$ and the existence of $Type_C$ both are redundant, since all variables and clocks are automatically associated with one or more UPPAAL process (namely those who use the variable or clock in assignments, guards or invariants), and each process is already associated with a RCX or the environment by $Type_A$. However, including the names in $Type_V$ and requiring the existence of $Type_C$ is convenient for implementation reasons.

4.2 Definition of the translation

In this section we define the translation from a UPPAAL model to RCX byte code program(s), in section 4.2.1, and to an UPPAAL model of the run-time behavior of the byte code, in section 4.2.2. Moreover, in section 4.2.3 we discuss the relation between the conceptual design and the generated byte code and model of that byte code. Finally, in section 4.2.4 we discuss our translation and the restrictions it imposes. We assume that the UPPAAL model has type mappings as explained in the previous section.

4.2.1 Generating the byte code program

This section describes how we translate a given UPPAAL model to one or more RCX byte code programs. We assume that the model consists of the set of processes A. Using the type mappings, we first determine which processes must be used for which byte code program:

$$R_n = \{\, a \in A \mid Type_A(a) = (rcx, n) \,\}$$

The set R_n contains all control program processes that must be combined to RCX byte code program with name n. The sets R_n that are not empty can be straightforward computed from the model.

All processes in a set R_n must be compiled into one byte code program. If all resulting byte code programs are run simultaneously on the RCX bricks, then the behavior of this LEGO system should ideally resemble the behavior of the original UPPAAL model.

We explain our translation in a top-down manner, using pseudo code instead of actual byte code. This is easier to understand and it is straightforward translatable to byte code. We start with an explanation of the main control structure of the byte code programs that simulates the interleaving of the various processes. Next, we explain how we translate the control structure of each process. Finally, we explain our translation of the edges that appear in the processes.

The main control structure

All processes in the set R_n are control program processes that must be combined to one byte code program. The semantics of UPPAAL interleaves the execution of all processes in the model in all possible ways. This behavior should as close as possible be simulated in the byte code program. However, we must choose one specific interleaving of the processes in R_n, since the scheduler of the firmware and the byte code language are deterministic. We distinguish two possibilities for the main control structure of the byte code program:

- A dedicated task for each process. The scheduler of the firmware takes care of interleaving the processes. One disadvantage is that a complicated mechanism is necessary to guarantee the atomicity of assignments on edges, which we want to preserve. Another disadvantage is that the number of control program processes for one RCX is bounded, since the firmware can handle at most 10 tasks.
- One large task that contains all processes. The scheduling is done by the control structure of this task. A disadvantage is that the processor is used less efficiently.

We choose for the second option, since we want atomicity of edges, but we do not want to complicate matters. Another reason to use one large task is that we do not need to construct a separate process for the scheduler of the firmware in the model of the run-time behavior of the byte code. The advantage of this will become clear in the next chapter. As a result, we use the main control structure for the byte code program as depicted in figure 4.1. As can be seen in the pseudo code, first the regular variables, sensors

```
task main()
{
    /* Declarations and initializations of program counters,
       variables, sensors and actuators. */

    while(1)
    {
        /* Translation of process 0 */

        /* Translation of process 1 */

        :

        /* Translation of process n */
    }
}
```

Figure 4.1: The main control structure.

and actuators that appear in the processes in R_n are declared and initialized. Moreover, we declare and initialize $|R_n|$ *program counters*, that are used to translate the control structures of the individual processes in R_n. Next, an infinite while loop is started that schedules the processes in a fixed order.

As we will see below, the atomic units of execution of the processes will be action and – implicitly – delay transitions, as is the case in UPPAAL.

Translation of the processes

For each process we have to keep track of the current location. This is done by a program counter, which is declared and initialized to the initial location before the infinite while loop. We construct one large if-then-else structure in order to simulate the execution of the process, as depicted in figure 4.2.

```
if (pc_P == 0)
{
    /* Translation of edges from location 0 */
}
else if (pc_P == 1)
{
    /* Translation of edges from location 1 */
}
:
else
{
    /* Translation of edges from location n */
}
```

Figure 4.2: Simulation of the execution of a process.

The variable `pc_P` is a program counter that is compared to integers that represent the locations of the process. If this structure has been added for every process, then the action transitions of the processes of R_n are interleaved in a fixed order. First process 0 can (try to) execute an action transition, then process 1 can attempt this, etcetera. Note that there is always an implicit time delay between the execution of edges due to the overhead of the control structure.

The possible location invariants in the control program processes are ignored by the translation. Location invariants are of the form $x \leq c$ where x is a clock and c is a constant. They ensure progress: control can only remain in the location for a bounded amount of time. As we will see in our explanation of the translation of edges below, the generated byte code *approximates* this progress automatically.

We also ignore the possible urgency or commitment of a location in the generated byte code. The semantics of urgency – if control is in an urgent location, then no time may elapse – are expressible with an extra clock and location invariant. As explained above, this urgency thus ensures progress, which is automatically approximated by the generated byte code.

The reason why we do not translate the semantics of commitment is that this would involve a complicated mechanism: different RCX bricks must let each other know whether or not some control program process is in a committed location. For example, if some process on RCX 1 is in a committed location, then a process on RCX 2 may only execute an action transition if it also is in a committed location.

Translation of edges

Above we explained how we simulate the interleaving of various processes by an infinite while loop that contains an if-then-else structure for every process. Now we explain the translation of the outgoing edges of each location to replace the comments in figure 4.2. Again, we use a large if-then-else structure to translate all outgoing edges for a given location. We distinguish two situations concerning the synchronization of the edge.

First, let us consider the situation where an edge *does not* synchronize with another process in R_n. Then, we use a scheme as depicted in figure

```
(else) if (guard)
{
    update pc_i
    assignments
}
```

Figure 4.3: Translation of an edge without synchronization.

```
(else) if (guard_i && pc_j==n && guard_j)
{
    update pc_i
    update pc_j
    assignments ! side
    assignments ? side
}
```

Figure 4.4: Translation of an edge with synchronization.

4.3. We use the (translation of) the guard of the edge as the guard of our if-case. The body of the if-case consists of the (translation of) the assignments on the edge and of an update of the program counter, to reflect a possible location switch. Of course, if a synchronization is present, then this must be taken into account. There are two possibilities:

- Synchronization with a process on another RCX brick. This is excluded by our translation. A sensible translation would involve much overhead and negotiation between different RCX bricks using the infrared channel.
- Synchronization with an environmental process. In this case, we mark the last byte code instruction that results from translation of the assignments of the edge with the synchronization label. We use this label for the generation of the model of the run-time behavior of the byte code, described in section 4.2.2.

Second, let us consider the the situation where an edge i *does* synchronize with an edge j of another process in R_n. Then, we use a scheme as depicted in figure 4.4. We check if both guards of the edges are satisfied, *and* if the other process is in the right location (denoted by `pc_j==n`). If this is true, then the synchronization can occur and the assignments are executed in the order that UPPAAL specifies. Finally, the program counters of both processes are updated.

Note that this scheme introduces a fixed preference for edges. If two outgoing edges of some location are enabled, then the edge that appears first in the large if-then-else structure, of which one element is depicted in the figures 4.3 and 4.4, is taken. Besides, this scheme ensures that an edge is taken if it is enabled, thus *as soon as possible.* This ensurance of progress is the reason why we do not explicitly translate the location invariants and urgency of locations of control program processes.

The translation of the guards and assignments is straightforward, using the types of the variables. The firmware does not support arrays and our translation does not use a mechanism to convert arrays to normal variables. Therefore, the use of arrays by control program processes is not allowed. For details about the translation, we refer to the implementation, that is described in section 4.3.

4.2.2 Generating the Uppaal model of the run-time behavior

We partially follow the approach of Iversen et al for the generation of the Uppaal model for the run-time behavior of the generated byte code [IKL$^+$00]. They have a timed automaton model for each kind of instruction, and construct the Uppaal model of the program by "concatenation" of these individual instruction models. Moreover, they explicitly include a process that models the simple round robin scheduler of the firmware. This is necessary, since a program may consist of multiple tasks in their approach.

Modeling the individual instructions

In the previous section we explained that our translation results in byte code programs of exactly one task and no subroutines. This enables us to include the model of the scheduler into the model of the program.

The scheduler checks the ten tasks in a cyclic, or round-robin, fashion. If a task is active, then it may execute one byte code instruction before the scheduler goes to the next task. Logically, the scheduler introduces some time overhead: it must check if a task is active and it must do some internal work to proceed to the next task. Since our generated byte code programs consists of only one task, this time overhead can be implictely modeled in the duration of byte code instructions.

We model every byte code instruction by one location, in which control can remain for the duration of the instruction plus ten times the overhead of the scheduler. This reflects the fact that the scheduler checks the other nine tasks between two instructions of the only active task of the program. For example, consider figure 4.5 that models a set instruction that assigns the value 5 to the variable `a`. Actually, the instruction is modeled by the left location and the edge. The right location is part of the model of the next instruction. The clock `x` is used to model the duration of the instruction, including the overhead of the scheduler. The instruction itself takes at least `L1` and at most `U1` time units. Similarly, the overhead of the scheduler is at least `10*SL` and at most `10*SU` time units. Note that clock `x` is reset for the next instruction.

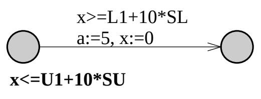


Figure 4.5: A model of the set instruction.

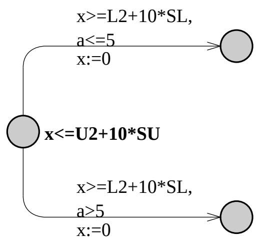


Figure 4.6: A model of the tbf instruction.

Figure 4.6 models a "test and branch far" instruction that is used to implement the if-then-else structures explained in the previous section. In this case, the test is whether or not variable a is less than or equal to 5. If this is true, then control is transferred to an instruction whose address is given in the tbf instruction. Otherwise, control is transferred to the next instruction. Note that the tbf instruction probably has another duration, expressed by L2 and U2, than the set instruction, but the same scheduler overhead.

Construction of the complete model

Let us consider an Uppaal model whose processes can be divided into the sets R_1 to R_n and E. The set R_i contains the processes that constitute the byte code program i, and the set E contains the environmental processes. In section 4.2.1 we explained how we can compile the sets $R_1, .., R_n$ to byte code programs, denoted by $B_1, .., B_n$. This section explains how we can generate *byte code processes* $P_1, .., P_n$ from the byte code programs. The generated model contains these byte code processes *and* the environment E of the source model.

We construct a new Uppaal process P_i for each byte code program B_i by concatenation of the individual models of the byte code instructions. The resulting processes must then be "plugged back" into the environment, that is given by the set E of the original (or source) Uppaal model. This is easy, except for the fact that the processes in R_i might synchronize with processes on other RCX bricks, or with environmental processes, as described in section 4.2.1 on page 45. We excluded the first case, but in the second case we attached the synchronization label, say $(a, !)$, to the last instruction that results from the translation of the assignment of the synchronizing edge in R_i. Thus an instruction in B_i has been labeled with $(a, !)$.

The translation of such a labeled instruction is the same as described above, except that we now also label the edge(s) of the instruction model with the synchronization label. For example, the edge of the set instruction model, depicted in figure 4.5, is labeled with $(a, !)$, if this instruction is the last instruction that results from translation of an edge labeled with $(a, !)$. As a result, this label (or channel) may not be declared as urgent, since there certainly is a clock guard present on the edge.

This scheme assures that the model of the byte code approaches the model of the conceptual design in its behavior with respect to synchronization with the environment. To illustrate the use of this, consider the conceptual design of the communication protocol as described in section 3.1 on page 17. The lossy infrared channels (the air) between the two RCXs in the system are modeled by two environmental processes. The control program processes "start" these channels by a synchronization. After the translation, the byte code processes must of course still be able to use these channels, and therefore the synchronization labels in the conceptual control program processes are transferred to the model of the run-time behavior of the byte code.

Translation of properties of the conceptual design

A conceptual design should, in general, satisfy some validation properties. When UPPAAL is used as the design tool, then these properties are invariance and/or reachability properties over state formulas, as described in section 2.2.1. These state properties can contain names of locations, clocks and variables. Thus, a problem arises when the properties of the original model are tested in the model of the generated byte code, since, e.g., the location names of control program automata are replaced by guards on program counters.

Another practical issue concerns the local objects of control program processes. Due to their locality, two processes in R_i may both have a local clock called `x`. When these two processes must be compiled into one byte code program, this may cause difficulties. A logical choice is to make these local objects global by replacing them by unique and global equivalents. Again, this influences the correctness of the original properties with respect to the generated model.

To overcome the sketched problems, we should also translate the original properties. As for the location names of control program processes, this is not difficult. Consider a control program process C and one of its locations L. Every appearance of this location, say `C.L`, in the original properties is replaced by the expression `pc_C==n`, where `pc_C` is the program counter of C and n is the integer that represents location L.

To translate the references to the local objects of the conceptual design in the original properties, we should of course know the renaming of the local objects. If we know this, then every appearance of a local object p of control program process C, e.g., `C.p<=6`, is replaced by its renaming, e.g., `C___p<=6`. This scheme assures that the translated properties are syntactically correct with respect to the generated model of the byte code.

4.2.3 Relation between the input and output of the translation

The byte code that is generated from a conceptual design does not completely resemble the behavior of that design. This is due to the fact that the semantics of Uppaal is not completely realizable by physical machines.

Probably the best example of unrealizable semantics is that Uppaal allows action transitions "with infinite speed". However, the RCX can certainly never execute a byte code instruction in zero time. It is clear that our translation allows conceptual designs with control program processes that exhibit this behavior: we allow urgent locations and channels, and location invariants (that can say $x \leq 0$).

We have a good reason for allowing unrealizable conceptual designs. If we did not allow them, then the conceptual design of a byte code program should include all implementation details, like the durations of byte code instructions, the scheduler, etcetera. Thus, the conceptual design should be very close to the model of the behavior of the generated byte code. Of course, this invalidates the approach of a relative simple and abstract *conceptual* design.

Other concepts that are difficult to realize in practice are the concurrency and non-determinism of Uppaal. Due to the facts that the RCX has only one processor and that it cannot execute instructions in zero time, concurrency of control program processes on the same RCX cannot be realized. As explained in section 4.2.1, the generated byte code introduces priorities for the outgoing edges of each location. If two edges of a location are enabled at the same moment, then the byte code always executes a fixed edge. Moreover, the fact that every action transition is taken as soon as possible in the generated byte code, lets action transitions prevail over delay transitions.

Concluding, we allow idealized conceptual designs in which control program processes can execute action transitions infinitely fast and in which control program processes are completely concurrent. The generated byte code, however, is not infinitely fast and only approximates *one* execution of the conceptual design. The behavior of the generated byte code is modeled by the other product of our translation. We believe that this generated model is fairly accurate. However, good care should be taken with the translated properties. For example, multiple assignments on one edge of a conceptual design are translated to multiple byte code instructions, and thus to multiple edges in the generated model. As a result, properties which assume that the atomicity of edges in the conceptual design is preserved, are probably not satisfied by the model of the generated byte code.

4.2.4 Restrictions introduced by the translation

Our translation imposes a number of restrictions on Uppaal models that we describe next. First, there are the straightforward restrictions that come forth from the RCX firmware.

- There are strict bounds on the number of variables and clocks that are used by a set of control program processes, denoted by R_i. As can be read in section 2.1, the firmware supports up to 32 integer variables and 4 timers for one program. Thus, the set of processes in R_i may at most use these numbers of variables and clocks. Note that there are temporary variables needed for the computation of arithmetic expressions.

- The firmware does not support arrays and we noted that our translation does not convert arrays to normal variables. Therefore, the control program processes are not allowed to use arrays.

The second type of restriction has to do with the shared use of variables, clocks and channels between sets of control program processes, denoted by R_i and R_j, and between a control program process and the set of environmental processes, denoted by E.

- The processes in R_i are not allowed to share clocks or variables with the processes in R_j. Since our translation does not support a sensible translation of such sharing – it is ignored – the behavior of the generated byte code differs too much from the conceptual design. Note that this requirement is automatically satisfied by the inclusion of the target RCX in the type mappings $Type_V$ and $Type_C$.

- We assumed that the only communication between the control program processes and their environment is through the sensors and actuators. We refine this by requiring that environment processes may only use untyped variables, or variables with type *ir_buffer* or *sns_x_value*, where $x \in \{1, 2, 3\}$ as targets for assignments. They are allowed, however, to use all variables and clocks in their guards. Similarly, environment processes may only reset untyped clocks.

- Our translation of synchronizations, described in section 4.2.1 on page 45, does not support synchronizations between processes in R_i and R_j. Therefore, we forbid these synchronizations. However, synchronizations between a process in R_i and and a process in E are allowed. Our assumption about the communication between a control program and the environment, results in our requirement that the environment must be *input enabled* with respect to these synchronizations. Moreover, since the last instruction of the edge is labeled with the synchronization label, we require that the control program process always uses the ! side. Finally, we explained that the channel is declared as non-urgent in the model of the generated byte code. If it was already non-urgent, then this poses no problems. Otherwise, it can, but we prevent them by requiring that only control program processes may use this channel.

Summarizing this last point, a control program process C is allowed to synchronize with an environmental process E using channel a, if (i) C only uses the label $(a, !)$, (ii) every non-committed location of E is the source of an unguarded edge labeled with $(a, ?)$, and (iii) if a is urgent, then no edge of any other environmental process is labeled with $(a, !)$. The channel processes in the communication protocol, depicted in figure 3.1 on page 19, illustrates these three requirements. First, the sender and receiver only use the ! side. Second, the commitment of the non-initial location assures that in any state the possibility exists for a synchronization. Third, the channels `sendSR` and `sendRS` are urgent, but no environmental process uses them.

Finally, we require the absence of committed locations in control program processes, since the semantics of committed locations – if some process is in a committed location then no time may elapse and the next action transition involves a process that is in a committed location – is not expressed in the byte code by our translation.

4.3 The implementation

In the sections 4.2.1 and 4.2.2 we explained the translation of a Uppaal model, that is a conceptual design, to byte code that implements this design and to another Uppaal model of the run-time behavior of the generated byte code. We implemented this translation, resulting in the compiler `uppaal2rcx`, which is available at the web site of this thesis.

Figure 4.7 depicts the input and output of the compiler. The dashed box at the left contains the Uppaal model of the conceptual design: a `.xml` and a `.q` file. The `.xml` file must first be exported as a `.ta` file, using the *save as* option in Uppaal. The `.ta` and `.q` file can then serve as input to

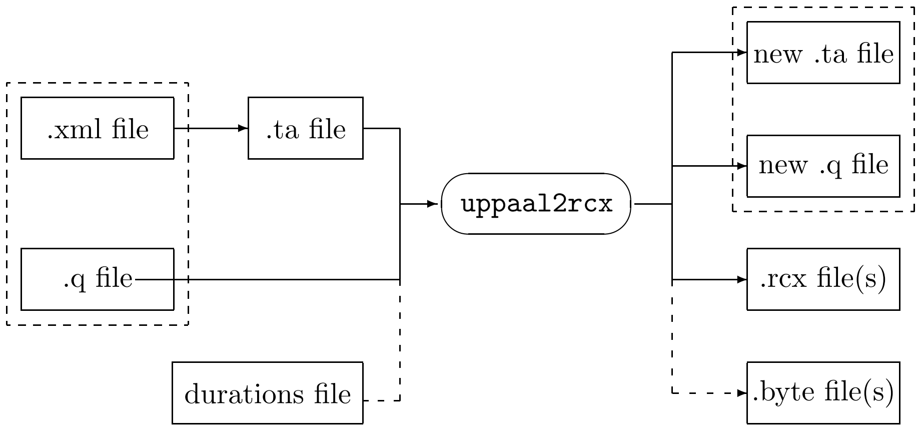


Figure 4.7: Input and output of `uppaal2rcx`.

the compiler. Moreover it is possible to specify the lower and upper bounds on the durations of the individual byte code instructions in a "duration file", which may also serve as input to the compiler. Due to the fact that the generated Uppaal model must contain these durations, which typically lie in the range of 10 to 100 microseconds, the compiler assumes a fixed time scale: one million UPPAAL time units model one second. Note that we were far-seeing during the conceptual design of the Level Crossing and the Production Cell: the time scale of these designs matches the time scale of the compiler.

If the model satisfies the requirements stated in section 4.2.4, then the compiler generates a new UPPAAL model, a set of `.rcx` files that are the actual byte code programs, and optionally a set of `.byte` files that contain symbolic byte code.

The reason why we use the `ta` file-format is that it is easier to parse and process than the `xml` file-format. This choice is not limiting since both the graphical front-end of and the stand-alone verifier of UPPAAL are capable of reading `ta` files.

Please note that we did also implement a download tool, called `rcxdownload`, which is available at the thesis' web site. This tool can be used to download the generated executable byte code to the RCXs using the IR tower of the LEGO Mindstorms set.

4.3.1 Adding the type mappings to a UPPAAL model

In section 4.1.2 we argued that extra information should be present in a UPPAAL model to facilitate the translation to executable code. We explained the type mappings that map variables, clocks and processes to the various RCX bricks in the system.

The type mappings can be added to the UPPAAL model by prefixing a declaration of a clock or variable with a comment. Consider for example the variable `sns2` that appears in the main controller of the train in figure 3.8 on page 27. The following combination of a comment and declaration of this variable maps the variable to the RCX with name `Gate` and sets its type to *sns_1_type*.

```
// RCX Gate sns_1_value
int sns1:=0;
```

Similarly, we can map clocks to RCXs in the following manner:

```
// RCX brick Gate
clock t;
```

This combination of a special comment with the declaration of clock `t`, maps the clock to the program for the RCX with name `Gate`. Finally, mapping the processes to RCXs or to the environment is done in the system description section of the UPPAAL model:

```
system

// RCX brick Train
TrainController, R,

// RCX environment
PhysicalTrain,
```

The first declaration maps the processes `TrainController` and `R` to the program for the RCX with name `Train`. The second declaration tells the compiler that the process `PhysicalTrain` is an environmental process, which should not be implemented.

If a model satisfies the requirements defined in section 4.2.4, then compilation of this model should result in byte code and a new UPPAAL model of the run-time behavior of the byte code. In the next section we give a small example of the translation.

4.3.2 A small example

To illustrate the translation we constructed a very small example for one RCX. Consider the processes P_0 and P_1 in figures 4.8 and 4.9. These processes model a reactive program, called B, which controls two actuators and uses two sensors. Process P_0 uses sensor 1 (whose value is modeled by the variable `in1`) and actuator A (whose mode is modeled by the variable `a`). Similarly, process P_1 uses sensor 2 and actuator B.

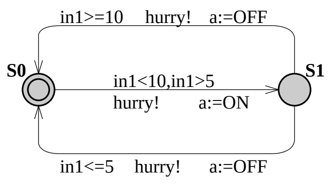


Figure 4.8: Process P_0.

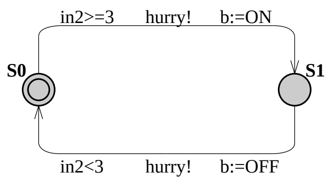


Figure 4.9: Process P_1.

Initially, both actuators are off. If the sensor value of sensor 1 becomes between 5 and 10, process P_0 switches actuator A on. If the sensor value leaves this region, then process P_0 switches actuator A off again. Process P_1 functions in a similar manner.

Figures 4.10 and 4.11 are the environmental processes. The hurry dummy provides an always enabled synchronization over the urgent channel `hurry`, which creates urgent edges. Note that all edges of P_0 and P_1 use this channel, with the result that they are taken as soon as possible. The environment periodically updates both sensor values with a “speed” expressed by the constant `LARGE`.

The type mappings are straightforward, and compilation of the model results in byte code that has been depicted as pseudo code in figure 4.12 and as symbolic byte code in figure 4.13. As the actual byte code is unreadable, we do not depict it here. Especially note the implementation of the main control structure and of the transitions.

Besides the byte code, the compiler also constructs a UPPAAL model of the byte code program in its environment. This model contains the hurry dummy and the environment as above, and the process of the actual byte

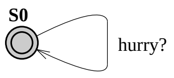


Figure 4.10: The hurry dummy.

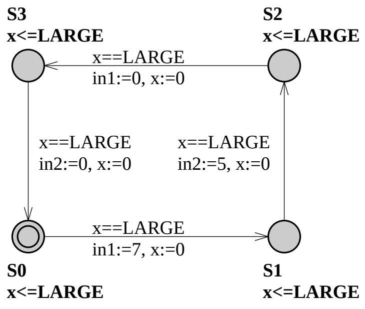


Figure 4.11: The environment.

code, depicted in figure 4.14 on page 56. There is a one-to-one mapping of locations to instructions: location `S`n maps to the $n + 1$-th instruction.

The keen reader probably has noted that the byte code process of figure 4.14 does not contain any synchronizations over the channel `hurry`, whereas its source processes, P_0 and P_1, use this channel on every edge. In section 4.2.2 on page 47 we explained that if some control program process synchronizes with the environment, then this synchronization is also present in the generated model. Following this scheme, the outgoing transitions of locations `S8`, `S14`, `S19`, `S26` and `S31` of the byte code process should have been labeled with `hurry!`.

Our implementation of the translation makes an exception for the case of synchronization over the channel `hurry`. It assumes that this channel is only used to create urgent edges. Since this urgency of edges is automatically (approximately) implemented by the byte code, these synchronizations are ignored by our compiler.

To motivate our choice of making an exception for the channel `hurry` in our translation of synchronization with the environment, consider the situation where we do not make the exception. Then the aforementioned edges in the byte code process would be labeled with `hurry!`, and this channel is necessarily declared as non-urgent, due to the clock guards one every edge of the byte code process. It is easy to see that this would not have any effect, but unnecessary overhead for the verification engine.

4.4 Experimental results

In this section we discuss experimental results of compiling the case studies. We discuss the behavior of the generated byte code on the physical LEGO setups, and the verification of the byte code using the generated models.

```
pc_P0 := 0;
pc_P1 := 0;
actmode[A] := off
actmode[B] := off

while (true)
{
    if (pc_P0 == 0)
    {
        if (snsval[0]<10 && snsval[0]>5)
        {
            pc_P0 := 1
            actmode[A] := on
        }
    }
    else
    {
        if (snsval[0]>=10)
        {
            pc_P0 := 0
            actmode[A] := off
        }
        else if (snsval[0]<=5)
        {
            pc_P0 := 0
            actmode[A] := off
        }
    }

    if (pc_P1 == 0)
    {
        if (snsval[1]>=3)
        {
            pc_P1 := 1
            actmode[B] := on
        }
    }
    else
    {
        if (snsval[1]<3)
        {
            pc_P1 := 0
            actmode[A] := off
        }
    }
}
```

Figure 4.12: Pseudo code for the translation of P_0 and P_1.

```
0000  v[0] := 0
0005  v[1] := 0
0010  actmode[A] := off
0012  actmode[B] := off


0014  tbf 0!=v[0], 51
0022  tbf 10<=snsval[0], 48
0030  tbf 5>=snsval[0], 48

0038  v[0] := 1
0043  actmode[A] := on
0045  baf 106
0048  baf 106


0051  tbf 10==snsval[0], 67
0059  tbf 10>=snsval[0], 77
0067  v[0] := 0
0072  actmode[A] := off
0074  baf 106
0077  tbf 5==snsval[0], 93
0085  tbf 5<=snsval[0], 103
0093  v[0] := 0
0098  actmode[A] := off
0100  baf 106
0103  baf 106

0106  tbf 0!=v[1], 143

0114  tbf 3==snsval[1], 130
0122  tbf 3>=snsval[1], 140
0130  v[1] := 1
0135  actmode[B] := on
0137  baf 164
0140  baf 164


0143  tbf 3<=snsval[1], 161

0151  v[1] := 0
0156  actmode[B] := off
0158  baf 164
0161  baf 164
0164  baf 14
```

Figure 4.13: Symbolic byte code for the translation of P_0 and P_1.

4.4.1 Testing the generated byte code of the Level Crossing

The conceptual design of the Level Crossing has been described in section 3.2. It consists of two files: `LC.xml` and `LC.q`. Using UPPAAL we can export the model in the `ta` format, thus creating the file `LC.ta`. Next, we invoke the compiler with the command `uppaal2rcx -s LC`. This results in six files:

- The files `LC-bytecode.ta` and `LC-bytecode.q` are the new UPPAAL model.

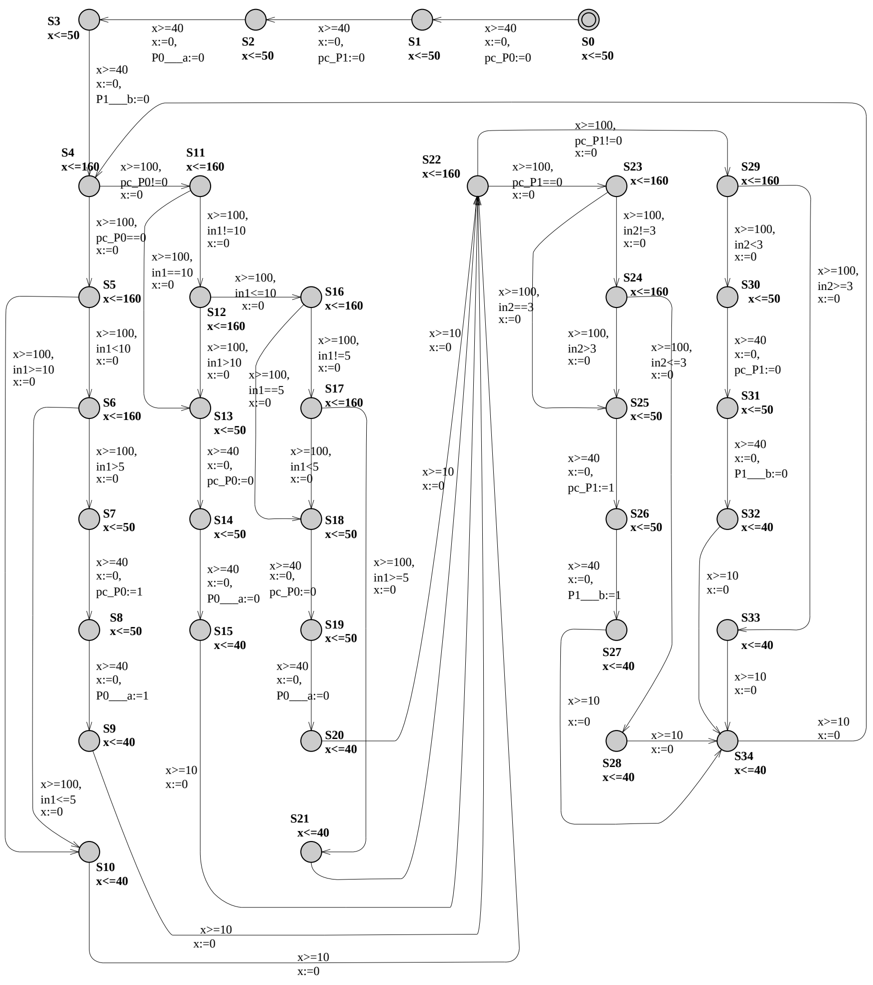


Figure 4.14: The UPPAAL process of byte code program *B*.

- The files `Train.rcx` and `Gate.rcx` are the executable byte code programs that can be downloaded to the RCXs using the tool `rcxdownload`. The file `Train.rcx` consists of 136 instructions which take 665 bytes. The file `Gate.rcx` is somewhat larger: it consists of 203 instructions which take 1060 bytes.

- The files `Train.byte` and `Gate.byte` contain the symbolic bytecode in ASCII format. They are useful for illustrative purposes and low-level debugging of the byte code.

It is noteworthy that the first few tests of the generated code for the Level Crossing, which used an earlier version of the compiler, were not successful. In this earlier version we used only one type, *ir*, for the infrared communication. A broadcast was modeled by assigning a value to the variable with type *ir*. At the same time, this variable modeled the local infrared buffer.

The first few tests of the generated byte code on the real LEGO setup failed when the communication protocol was started. After some low-level digging we found that the broadcast instruction of the RCX does not update the local infrared buffer, as we assumed. Thus, there was a fairly large gap between the conceptual design and the implementation. A broadcast also updated the local infrared buffer in our conceptual design. The generated byte code, however, did not do this. This detail caused the receiver to continuously send acknowledgements to the sender after the reception of the first message.

The strange behavior of the broadcast instruction made it necessary to introduce two variable types, *ir_buffer* and *ir_value*, to model the infrared communication between RCXs. This has been explained in section 4.1.2.

After adaptation of the compiler and the model of the communication protocol, resulting in the model described in section 3.1, we tested again. We tried all paths through the control graph of the main controller of the gate, depicted in figure 3.7 on page 25. The system functioned as we expected: all delays were approximately as specified in the conceptual design, and the appropriate actions were taken.

4.4.2 Testing the generated byte code of the Production Cell

The conceptual design of the Production Cell has been described in section 3.3. It consists of two files: `PC.xml` and `PC.q`. Using UPPAAL we can export the model in the `ta` format, thus creating the file `PC.ta`. Next, we invoke the compiler with the command `uppaal2rcx -s PC`. This results in six files:

- The files `PC-bytecode.ta` and `PC-bytecode.q` are the new UPPAAL model.
- The files `Beltcontroller.rcx` and `RobotarmController.rcx` are the executable byte code programs. The file `Beltcontroller.rcx` consists of 166 instructions which take 851 bytes of space. The second file, `RobotarmController.rcx`, is somewhat larger: it consists of 215 instructions which take 1044 bytes.
- The files `Beltcontroller.byte` and `RobotarmController.byte` contain the symbolic byte code in ASCII format.

The generated code for the belt controller functioned as expected. It provides the essential mutual exclusion of bricks on belt A, and it transports all arriving bricks to the end of belt A. The code for the robot arm controller, however, did not function as we expected. The cause(s) of this erroneous behaviour have not been discovered due to lack of time (the testing was conducted in Aalborg).

4.4.3 Verifying the Level Crossing

The model of the Level Crossing contains processes for the train controller and processes for the gate controller. To avoid large models, we map either the gate controller processes or the train controller processes to the environment. Application of the compiler then results in one byte code program and in a model that only contains one byte code process. The other control program processes are regarded as the environment and are the same as in the source model.

First, we mapped the control program processes of the gate controller to the environment. Thus, the process of the main controller of the train and the process of the receiver of the communication protocol are combined to one bytecode process.

We tried to verify the four properties which we stated with our conceptual design, see table 4.1. All runs of the model checker use a breadth-first search order. We use a threshold time of 4 hours to classify properties as practical verifiable or not. If we halted the model checking process, then there is a question mark in the "Satisfied?" field. The verdict "maybe" can appear when the convex-hull approximation is used. This approximation can only be used to verify the truth of invariance and untruth of reachability properties. For example, if an invariance property is "maybe" true, then the model checker found a counter example. However, without the approximation, the property still could hold.

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(3.3)	16 : 50 : 33	6	`-A`	?
(3.4)	19 : 55	3	`-A`	maybe
	5 : 42 : 49	105		?
(3.5)	19 : 54	3	`-A`	maybe
	20 : 54 : 11	287		?
(3.6)	6 : 46 : 35	5	`-A`	?

Table 4.1: Model checking the implementation of the train controller.

We first tried to check the properties using the convex-hull approximation (`-A` option), which is a safe over-approximation. If the model checking process terminated within four hours, and if it did not give an exact answer, then we also tried a run without the approximation.

The results are not very encouraging since we have no conclusive answers about the truth of our properties. From our results we can conclude that the reachable state space – even with the convex hull approximation – is too large to explore within 16 hours.

Table 4.2 shows the results for the model that is generated when only the processes of the gate controller are implemented. Again, the results show us that practical verification is at least very time consuming. Moreover, we do

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(3.3)	16 : 51 : 50	5	`-A`	?
(3.4)	26 : 28 5 : 57 : 30	3 93	`-A`	maybe ?
(3.5)	26 : 25 20 : 54 : 52	3 290	`-A`	maybe ?
(3.6)	32 : 47 5 : 56 : 53	3 92	`-A`	maybe ?

Table 4.2: Model checking the implementation of the gate controller.

not get any conclusive answers about the truth of our properties.

4.4.4 Verifying the Production Cell

We split the verification of the Production Cell in two parts as with the Level Crossing. First, we consider the implementation of the belt controller. Fortunately, the correct functioning of the belt controller does not depend heavily on the actions of the robot arm controller. Therefore, we remove the processes concerning the robot arm from the model. In order to verify that the belt controller brings the bricks to the pickup position, we introduce property (4.1).

$$\exists\Diamond(\texttt{Brick1.at\_sensor1\_safe}) \tag{4.1}$$

The second measure to make the model as small as possible is to remove a brick from the system, leaving one brick in the system. The only useful property that remains from those stated with the conceptual design, is property (3.10). Table 4.3 shows that we cannot obtain any conclusive answers

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(4.1)	18 4 : 52 : 35	3 97	`-A`	maybe ?
(3.10)	27 4 : 27 : 17	3 93	`-A`	maybe ?

Table 4.3: Model checking the implementation of the belt controller.

about the truth of these two properties.

In order to check the implementation of the robot arm controller, we need the belt controller, since it must transport the brick to the end of belt A, and signal the robot arm controller that the brick is ready. Therefore, we map the processes that compose the belt controller to the environment. Again, we leave only one brick in the system. Table 4.4 shows that we were not able to obtain conclusive answers about the truth of our properties.

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(3.8)	5 : 17 : 42	17	`-A`	?
(3.9)	5 : 17 : 34	17	`-A`	?
(3.10)	5 : 17 : 36	14	`-A`	?
(3.13)	2 : 39	5	`-A`	maybe
	5 : 08 : 27	282		?

Table 4.4: Model checking the implementation of the robot arm controller.

4.4.5 Discussion

It seems that the practical use of models of the byte code is neglectable, since model checking takes very much time. We believe that the main reason for the large symbolic state spaces is the *difference in time scale* between the byte code process and the environment. The durations of the byte code instructions lie in the range of 10 to 200 microseconds, whereas the delays in the environment are often specified in terms of seconds.

The timed automaton of example 2.1 on page 11 might help to explain the effect of different time scales. This automaton can be regarded as a parallel composition between a control program process and the environment. The cycle `L0`, `L1`, `L2` corresponds to cyclic execution of a control program consisting of three atomic instructions with the invariants and guards on the clock `y` providing execution time information. Whenever the control cycle is in location `L0`, the environment (modeled by the clock `z`) is consulted potentially leading to an exit of the control cycle. The size of the threshold constant `LARGE` determines how slow the environment is relative to the execution time of control program instructions: the larger the constant the slower.

Table 2.1 on page 14 shows the symbolic states of the first execution of the cycle. It is clear that every new execution of the cycle gives rise to new symbolic states, until the values of the clock(s) exceed the largest constant in the model. Therefore, the number of reachable symbolic states increases with the value of `LARGE`, which we call the *fragmentation problem.*

A similar fragmentation of the reachable symbolic state space occurs with our Level Crossing and Production Cell. The byte code processes display busy-waiting behavior: control just cycles through the various “test and branch” and “branch always far” instructions until a guard is satisfied and an action transition can be executed. This unnecessary cycling is very fast in comparison with the environment and fragments the symbolic state space in a dramatic way, as can be seen in the previous sections with experimental results.

4.5 Summary

This large chapter described the heart of our technique for obtaining correct reactive programs for the RCX using the model checker Uppaal. We defined a translation from Uppaal models to RCX byte code and a new Uppaal model of the run-time behavior of that byte code.

We argued that a bare Uppaal model does not contain enough information to implement it on an arbitrary hardware platform. Before one can think of implementation, one must know which processes are the control program processes and which are the environmental process. Moreover, for each variable in a control program process, one must know whether it models a sensor value, or a sensor mode, etcetera. These matters of distribution and mapping of implementation details have been solved by annotating the Uppaal model with three *type mappings*.

Next, we explained how we generate executable byte code from a Uppaal model with type mappings. We discussed the main control structure and the interleaving of the various control program processes. Due to the physical abilities of the RCX and the nature of the firmware, concepts like *concurrency* and *non-determinism*, which are part of Uppaal's semantics, are lost in the byte code.

In the same section, we explained how we can easily construct a fairly accurate Uppaal model of the run-time behavior of the generated byte code. Moreover, we have shown how the verification properties of the source model can be translated in such a way that they are syntactically correct for the generated model. The semantics of the generated properties lie very close to those of the source properties.

We concluded the explanation of the translation with some requirements that conceptual designs should satisfy. First, the conceptual design should "fit" on the target platform. For example, the firmware of the RCX can only use 3 sensors; a conceptual design should respect this. The second type of requirements originates from the need to keep the implementation close to the conceptual design. For example, processes mapped to different RCX bricks are not allowed to synchronize, since our translation does not implement this.

We implemented the translation and the tool – `uppaal2rcx` – is available at the web site of this master's thesis. In order to use the tool, the source Uppaal model should be annotated with special comments that define the type mappings. We explained how this is done, and we compiled a very small theoretical example to show the form of the output of the compiler.

Finally, we began with the real thing and applied our compiler to the conceptual designs of the Level Crossing and Production Cell. The generated byte code for the Level Crossing functioned, after a slight adjustment of the compiler, as expected. Part of the generated code for the Production Cell, however, did not function well. Due to lack of time we did not track the causes for the erroneous behavior. It is noteworthy that the generated byte code is fairly complex. Writing it manually is not a very pleasant, and certainly a time consuming task.

Model checking the generated models proved to be practically impossible. We were unable to obtain decisive answers about the truth of our verification properties. We concluded that a main reason is the *difference in time scale* between the byte code processes and the environmental processes. We address this problem in the next chapter.

Chapter 5

Exact acceleration of real-time model checking

An important problem concerning symbolic model checking of timed automata, is encountered when the timed automata in a model use different *time scales*. This, for example, is often the case for models of reactive programs with their environment. Typically, the automata that model the reactive programs are based on microseconds whereas the automata of the environment function in the order of seconds. This difference can give rise to an unnecessary fragmentation of the symbolic state space. As a result, the time and memory consumption of the model check process increases.

The fragmentation problem has already been encountered and described by Hune and Iversen et al during the verification of LEGO Mindstorms programs using UPPAAL [Hun00, IKL$^+$00]. The symbolic state space is severely fragmented by the busy-waiting behavior of the control program automata. This problem can in general occur during the symbolic model checking of systems that are modeled by timed automata. Examples include the aforementioned reactive programs, and polling real-time systems, e.g., programmable logic controllers [Die99]. The validation of communication protocols will probably also suffer from the fragmentation problem when the context of the protocol is taken into account.

In this chapter we propose an acceleration technique for a subset of timed automata, namely those that contain special cycles, that addresses the fragmentation problem. Our technique consists of a syntactical adjustment that can easily be computed from the timed automaton itself. We prove that this syntactical adjustment is *exact* with respect to reachability properties *and* that it can effectively speed-up forward symbolic reachability analysis. As a result, our approach is readily applicable using the existing model checkers. We demonstrate the exact acceleration by experimental results obtained with a theoretical example using the model checkers UPPAAL and KRONOS.

Unfortunately, it appears that our acceleration technique loses its exactness when it is generalized to UPPAAL models. However, our technique still provides an over-approximation. We explain the automatic application of this over-approximating acceleration, and apply it to the Level Crossing and the Production Cell. Fortunately, we obtain better verification results than those in section 4.4; we were able to verify some of the invariance properties within reasonable time.

Related work. Closely related work has been done in the field of symbolic verification of systems that are modeled by a discrete control graph with unbounded integer variables [BW94]. Static analysis of the control graph is used to detect *interesting cycles*, of which the result of iterated execution can be computed by one single *meta transition.* These meta transitions are then added to the system and favored by the state space exploration algorithm, resulting in faster exploration of the state space.

Symbolic techniques using *queue-content decision diagrams*, or QDDs, for the analysis of communication protocols that are modeled by finite-state machines that communicate through unbounded FIFO-queues, also use meta transitions to accelerate the exploration of the state space [BG96, BGWW97]. Special cycles in the control-graph, e.g., the repeated receiving of messages from a channel, are associated with meta transitions that compute all states that are reachable by the iterative execution of the cycle. In these approaches only a limited class of cycles in the control graph can be accelerated due to the expressivity of QDDs. To overcome this problem, *constrained* QDDs have been introduced, that allow the acceleration of *any* cycle in a control graph [BH97].

Recently, acceleration techniques have been proposed in the setting of parameterized model checking [ABJN99, PS00]. The techniques, again, compute the effect of an unbounded number of actions to accelerate the forward exploration process.

Möller's "parking" approach to the sketched fragmentation problem is, like our approach, based on a syntactical adjustment of timed automata to speed-up the state space exploration [Möl02]. The parking idea is more general than ours, but our method is *exact*, whereas parking is mostly an over-approximation. Möller applies his approach to an example somewhat larger than our examples, and measures speed-ups in the same order of magnitude as we do. We think that both methods show promises for handling the fragmentation problem.

In a sense, the syntactical adjustment of our approach also is a meta transition that computes the result of iterated execution of a cycle in the timed automaton. Using a breadth-first search order then guarantees that the exploration of this meta transition is not postponed. As far as we know this is the first application of acceleration techniques to timed automata.

Outline. In section 5.1 we first give a toy example that illustrates why different time scales in models can increase the reachable symbolic state space. Moreoever, we define our syntactic adjustment and we prove that it is exact with respect to reachability and that it indeed is effective. The proofs of the lemmas and theorems which appear are enclosed in appendix A. Finally, we illustrate our main results by experimental data.

In section 5.2 we discuss how our technique can be applied to general

Uppaal models. We show that our technique mostly loses its exactness, but that it always remains an over-approximation. Finally, we explain how we enhanced our compiler `uppaal2rcx` with over-approximating acceleration and we demonstrate its use by applying it to the Level Crossing and Production Cell.

5.1 Exact acceleration

The timed automaton of example 2.1 on page 11, depicted again below, illustrates the fragmentation of the reachable symbolic state space. It offers

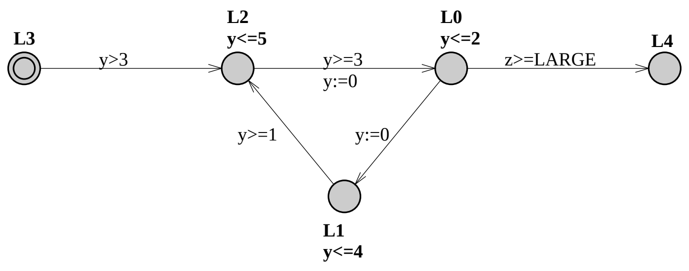


Figure 5.1: Automaton P.

a simplified modeling of a control program combined with an environment. The cycle `L0`, `L1`, `L2` corresponds to cyclic execution of a control program consisting of three atomic instructions with the invariants and guards on the clock `y` providing execution time information. Whenever the control cycle is in location `L0`, the environment (modeled by the clock `z`) is consulted potentially leading to an exit of the control cycle. The size of the threshold constant `LARGE` determines how slow the environment is relative to the execution time of control program instructions: the larger the constant the slower. The table 2.1 on page 14 shows the symbolic states of the first execution of the cycle. It is clear that every new execution of the cycle gives rise to new symbolic states, until the values of the clock(s) exceed the largest constant in the model. Therefore, the number of reachable symbolic states increases with the value of `LARGE`, which we call the fragmentation problem.

In this section we propose a technique that eliminates the fragmentation that is due to a special kind of cycles. In section 5.1.1 we give some basic definitions concerning cycles in timed automata. The subset of cycles that we can accelerate, is defined in section 5.1.2. Finally, in section 5.1.3 we define the syntactical adjustment of a subset of timed automata that accelerates the forward symbolic reachability analysis. We prove that this adjustment is exact with respect to reachability properties and that it accelerates the forward symbolic exploration of the reachable state space.

5.1.1 An introduction to cycles

We start this section with the definition of two functions to obtain the source and target locations of an edge of a timed automaton.

$$src((l, a, \phi, \lambda, l')) = l$$
$$trg((l, a, \phi, \lambda, l')) = l'$$

For a sequence of edges $E_c = (e_0, e_1, ..., e_{n-1}) \in E^n$ of a timed automaton, we let $Loc(E_c)$ denote the set of locations that appear in the edges:

$$Loc(E_c) \quad = \quad \{\, l \in L \,|\, \exists_{e \in E_c} [\, src(e) = l \,\vee\, trg(e) = l \,]\,\}$$

A cycle in a timed automaton is a sequence of edges, defined as follows:

Definition 5.1 (Cycle) *Let $M = (L, l^0, \Sigma, X, I, E)$ be a timed automaton and let $n \geq 1$. We say that a sequence $(e_0, e_1, ..., e_{n-1}) \in E^n$ is a* cycle*, if the following holds:*

- *$trg(e_i) = src(e_{i+1})$ for all $0 \leq i < n-1$, and $trg(e_{n-1}) = src(e_0)$, and*
- *$i \neq j \Rightarrow e_i \neq e_j$ for all $0 \leq i, j < n$.*

The timed automaton of figure 5.1 contains a cycle that, for example, is defined by the edges `L0` to `L1`, `L1` to `L2` and `L2` to `L0`. For a cycle, we can define the number of times that it is executable.

Definition 5.2 (Cycle execution) *Let $E_c = (e_0, e_1, ..., e_{n-1})$ be a cycle in some timed automaton M and let $m > 0$. We say that the cycle is m-times executable, if a finite compressed trace in $Tr(M)$exists with a suffix, say of the form*

$$((l_0, \nu_0), (l_0, \nu_0'), (l_1, \nu_1), ..., (l_{k-1}, \nu_{k-1}'), (l_k, \nu_k))$$

where $l_0 = src(e_0)$, such that the following holds:

- *the i-th action transition $((l_i, \nu_i'), (l_{i+1}, \nu_{i+1}))$ corresponds to the edge $e_{i \bmod n}$, and*
- *there are $m \cdot n$ action transitions.*

Example 5.1 *The cycle in the timed automaton of figure 5.1 is 1-time executable. This can be understood from the following suffix of a finite compressed trace, of which the first state obviously is reachable from the initial state (we denote the clock interpretation ν by a tuple that first contains the value $\nu(\mathtt{y})$ and second the value $\nu(\mathtt{z})$):*

$$\Big((\mathtt{L0},(0,4)), (\mathtt{L0},(1,5)), (\mathtt{L1},(0,5)), (\mathtt{L1},(2,7)), (\mathtt{L2},(2,7)), (\mathtt{L2},(4,9)), (\mathtt{L0},(0,9))\Big)$$

Cycles in timed automata suffer in general from a certain delay due to invariants at locations and clock guards on edges. In many cases, this delay varies for every execution of the cycle. However, we will later see that there exist cycles with a “fixed” window of delay for each execution of the cycle. But first, we define this window as an interval containing all possible delays that can be accumulated by following the cycle exactly once.

Definition 5.3 (Window of a cycle) *Let us consider a timed automaton M and let $E_c = (e_0, ..., e_{n-1})$ be a cycle in M. We say that an interval $[a, b]$ is the window of E_c, if for all subsequences of compressed traces in $Tr(M)$, say of the form*

$$((l_0, \nu_0), (l_0, \nu_0'), (l_1, \nu_1), ..., (l_{k-1}, \nu_{k-1}), (l_{k-1}, \nu_{k-1}'), (l_k, \nu_k))$$

such that $l_0 = l_k = src(e_0)$ and every action transition $((l_i, \nu_i'), (l_{i+1}, \nu_{i+1}))$ is due to edge e_i (this subsequence thus denotes exactly one execution of E_c), the following holds:

- *the total amount of delay in this subsequence is an element of $[a, b]$, and*
- *for all $d \in [a, b]$ it holds that we can adjust the delays in the subsequence such that they accumulate to d, and there exists a trace in $Tr(M)$ of which it is a subsequence.*

This window property is not trivial. We can prove that not every cycle has a window by providing a counter example to the statement that every cycle has a window.

Lemma 5.1 *Not every cycle has a window.*

Moreover, there exist cycles that do have a window, as we will see in lemma 5.3.

5.1.2 Acceleratable cycles

In this section we introduce a subset of interesting cycles in timed automata. These interesting cycles can use only one clock in the invariants, guards and resets. This clock can be used to specify lower and upper bounds on the edges of the cycle. This might seem like a strong restriction, but we argue that these kind of cycles occur often in control graphs of, e.g., polling real-time systems.

Definition 5.4 (Acceleratable cycle) *Let $M = (L, l^0, \Sigma, X, I, E)$ be a timed automaton, let $E_c = (e_0, ..., e_{n-1}) \in E^n$ and let $y \in X$. We say that the tuple (E_c, y) is an acceleratable cycle, if*

- *E_c is a cycle,*

- *$I(l)$ is empty or has the form $\{y \leq c\}$ for all $l \in Loc(E_c)$,*
- *if $(l, a, \phi, \lambda, l') \in E_c$, then either ϕ is empty or has the form $\{y \geq c\}$, and λ is empty or only contains y, and*
- *y is reset on all ingoing edges to $src(e_0)$.*

Clock y is called *the clock of the cycle* and location $src(e_0)$ is called *the reset location* from now on. The cycle in our example automaton is an acceleratable cycle, if we choose clock `y` as the clock of the cycle and `L0` as the reset location. The guards and invariants have the correct form for clock `y` and this clock is reset on the only incoming edge of `L0`.

To extract the constants from the clock guards and invariants, we define two partial functions cn_g and cn_I that map clock guards and invariants to natural numbers:

$$\begin{array}{ll} cn_g(\phi) = 0 & \text{if } \phi = \emptyset \\ cn_g(\phi) = c & \text{if } \phi = \{y \geq c\} \\ \\ cn_I(\phi) = \infty & \text{if } \phi = \emptyset \\ cn_I(\phi) = c & \text{if } \phi = \{y \leq c\} \end{array}$$

Acceleratable cycles have the property that if the cycle can be executed once, then it can be executed infinitely often. Consider, for example, the finite compressed trace of example 5.1. The first and the last state of this trace agree on the value of clock `y`. Since the guards and invariants in the cycle are solely concerned with this clock, the sequence action and delay transitions can be repeated an arbitrary number of times.

Lemma 5.2 (Cycle consecution) *Let (E_c, y) be an acceleratable cycle of some timed automaton M. If E_c is 1-time executable, then it is m-times executable, for all $m > 0$.*

Our acceleratable cycles have a window, that can be computed from the syntax of the timed automaton.

Lemma 5.3 (Window computation) *Each acceleratable cycle has a window.*

Sketch of proof. Let a timed automaton be defined by the tuple $M = (L, l^0, \Sigma, X, I, E)$ and let $((e_0, e_1, ..., e_{n-1}), y)$ be an acceleratable cycle. We will show that we can effectively compute the window from the syntax of the timed automaton.

Let $(l_i, a_i, \phi_i, \lambda_i, l_{i+1})$ denote edge e_i. We can find p natural numbers $0 \leq k_0 < k_1 < ... < k_{p-1}$ that exactly correspond to the indices of the edges on which clock y is reset. Since we know by definition that y is reset on edge e_{n-1}, p is at least one. Next, we compute the following numbers for $0 \leq j < p$ (we define $k_{-1} = -1$):

$$\begin{array}{lcl} a_{k_j} & = & max\,\{\, cn_g(\phi_i) \mid k_{j-1} < i \leq k_j \,\} \\ b_{k_j} & = & cn_I(I(l_{k_j})) \end{array}$$

Since we consider the guards and invariants of an acceleratable cycle, all the numbers a_{k_j} and b_{k_j} are defined. We can show that the acceleratable cycle has a window of

$$\left[\sum_{j=0}^{p-1} a_{k_j}\, , \sum_{j=0}^{p-1} b_{k_j}\right]$$

□

Example 5.2 *We saw that the timed automaton of figure 5.1 has an acceleratable cycle starting in* `L0`. *Applying our technique of window computation, we first obtain that* e_0 *is the edge from* `L0` *to* `L1`, e_1 *is the edge from* `L1` *to* `L2`, *and* e_2 *is the edge from* `L2` *to* `L0`. *Since clock* `y` *is reset two times on these edges, we have* $p = 2$ *and* $k_0 = 0$ *and* $k_1 = 2$. *Thus,* $a_0 = max\{0\} = 0$ *and* $a_1 = max\{1, 3\} = 3$. *Moreover,* $b_0 = cn_I(I(\texttt{L0})) = 2$, *and* $b_1 = cn_I(I(\texttt{L2})) = 5$. *Therefore, the acceleratable cycle has a window of* $[3, 7]$.

5.1.3 Acceleration

The motivation of this chapter is the acceleration of real-time model checking. We explained that the time that model checking of the property $\exists\Diamond(\texttt{L4})$ for the timed automaton of figure 5.1 takes, is very dependent on the value of the constant `LARGE`. This is due to the fact that many executions of the cycle must be explored to let the value of clock `z` grow large enough. The following definition appends an extra cycle, the meta transition, to automata with an acceleratable cycle. As we will see, this appended cycle computes the effect of the iterated execution of the acceleratable cycle.

Definition 5.5 (Acceleration of timed automata) *Let the tuple* $M = (L, l^0, \Sigma, X, I, E)$ *be a timed automaton and let* $A = ((e_0, ..., e_{n-1}), y)$ *be an acceleratable cycle. Let* $L = \{l_0, l_1, ..., l_m\}$, *and let* $e_i = (l_i, a_i, \phi_i, \lambda_i, l_{i+1})$. *The acceleration of* M *is a new timed automaton* $Acc(M, A)$, *defined by the tuple* $(L_{new}, l^0, \Sigma, X, I_{new}, E_{new})$, *where*

$$L_{new} = L \cup \{l'_1, l'_2, ..., l'_{n-1}\} \cup \{l'_0\} \cup \{l''_1, l''_2, ..., l''_{n-1}\}$$

$$\begin{aligned}
I_{new}(l_i) &= I(l_i) \textit{ for all } 0 \le i \le m\\
I_{new}(l'_i) &= I(l_i) \textit{ for all } 1 \le i \le n-1\\
I_{new}(l'_0) &= \emptyset\\
I_{new}(l''_i) &= I(l_i) \textit{ for all } 1 \le i \le n-1
\end{aligned}$$

$$\begin{aligned}
E_{new} = E \;\cup\; & \{\, (l_0, a_0, \phi_0, \lambda_0, l'_1), (l'_{n-1}, a_{n-1}, \phi_{n-1}, \lambda_{n-1}, l'_0) \,\} \cup\\
& \{\, (l'_0, a_0, \phi_0, \lambda_0, l''_1), (l''_{n-1}, a_{n-1}, \phi_{n-1}, \lambda_{n-1}, l_0) \,\} \cup\\
& \{\, (l'_i, a_i, \phi_i, \lambda_i, l'_{i+1}), (l''_i, a_i, \phi_i, \lambda_i, l''_{i+1}) \,|\, \textit{for all } 1 \le i < n-1\}
\end{aligned}$$

Note that the definition of E_{new} *does not cover the simple case where* $n = 1$. *Then, only two edges must be added:* $(l_0, a, \phi, \lambda, l'_0)$ *and* $(l'_0, a, \phi, \lambda, l_0)$, *if* $e_0 = (l_0, a, \phi, \lambda, l_0)$.

Example 5.3 *Since the timed automaton of figure 5.1 has an acceleratable cycle, we can construct the acceleration, see figure 5.2. The key idea behind*

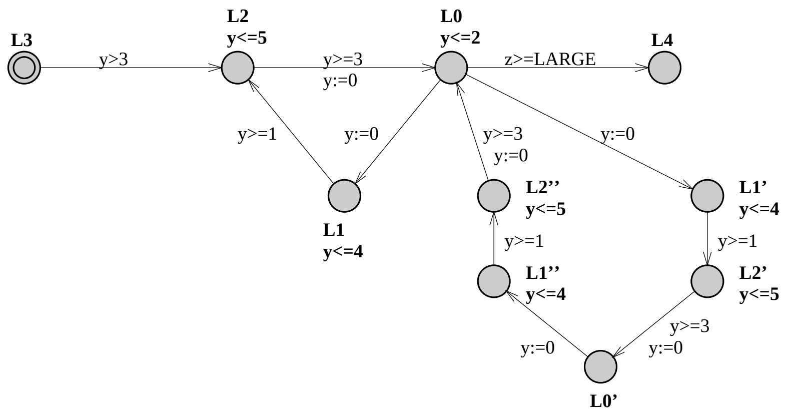


Figure 5.2: Automaton P_A: the accelerated version of P.

the acceleration is that the new location `L0'`*, that mimics location* `L0`*, has no invariant [Möl02].*

This acceleration is only interesting if we can use the accelerated version of some automaton to model check properties of the original automaton. Theorem 5.1 ensures this, provided that the window of the acceleratable cycle is relatively wide enough.

Theorem 5.1 (Equivalence of reachability) *Let* $(L, l^0, \Sigma, X, I, E)$ *be a timed automaton* M*, let* A *be an acceleratable cycle of* M *with a window of* $[a, b]$*, and let* ϕ *be a reachability properties of* M*.*

$$3a \leq 2b \Rightarrow (M \models \phi \Leftrightarrow Acc(M, A) \models \phi)$$

If the precondition of this theorem is *not* satisfied, then the acceleration is still a safe over-approximation. Thus, if a state is unreachable in $Acc(M, A)$, then it is also unreachable in M. The fact that definition 5.5 unfolds the acceleratable cycle twice to form the appended cycle, is the direct cause of the necessary relative width of the window. It can be shown that the precondition can be generalized to $(i + 1)a \leq ib$, where i is the number of unfoldings of the acceleratable cycle. This means that if a is strictly less than b, then we *can* accelerate. At this time we do not know how to handle the simple case where $a = b$.

It may seem that the addition of extra locations only increases the state space. However, we claim that if the clock of the acceleratable cycle is also reset on the first edge of the cycle, then the acceleration is guaranteed to work for breadth-first forward symbolic reachability analysis. Note that if a timed automaton has an acceleratable cycle, but does not satisfy the constraint described above, then we can introduce a dummy location to ensure that this requirement is satisfied.

Example 5.4 *Suppose that the timed automaton of figure 5.1 does not have a reset of* `y` *on the edge from* `L0` *to* `L1`*. Then we can construct the "equivalent" automaton depicted in figure 5.3. We added a dummy location between*

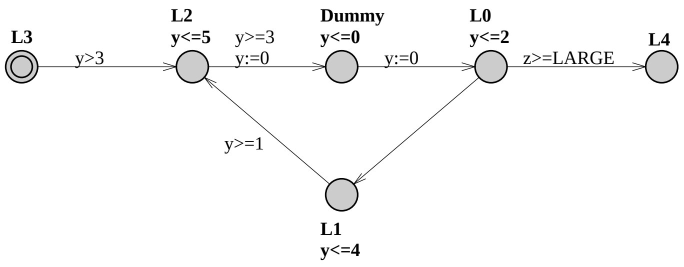


Figure 5.3: Adding a dummy location.

the locations `L2` *and* `L0` *with invariant* $\mathtt{y} \leq 0$*. Next, the edge from* `L2` *to* `L0` *has been redirected to the dummy location. Finally, we added an extra edge from the dummy location to location* `L0` *that resets clock* `y`*. We now can accelerate the cycle from the dummy location.*

The cycle that results from our trick is larger, but with the dummy location as reset location, it satisfies the requirement for the effectiveness of acceleration, as formalized in the next theorem.

Theorem 5.2 (Effectiveness of acceleration) *Let the timed automaton* M *be defined by* $(L, l^0, \Sigma, X, I, E)$ *and let* $A = ((e_0, ..., e_{n-1}), y)$ *be an acceleratable cycle. If* y *is reset on edge* e_0*, then all states reachable by more than one execution of the acceleratable cycle in* M*, are reachable by exactly one execution of the appended cycle in* $Acc(M, A)$*.*

This theorem guarantees us that breadth-first forward symbolic reachability analysis is accelerated. When using a breadth-first search-order, the appended cycle is "in parallel" explored with the first few executions of the acceleratable cycle. Our effectiveness theorem ensures that the symbolic state that results from the execution of the appended cycle swallows the symbolic states that result from two or more executions of the acceleratable cycle. Therefore, the acceleratable cycle is only explored a small number of times. (The exact number of times depends on the implementation of the model check algorithm, but it will be independent of the difference in time scale.)

5.1.4 Experimental results

To demonstrate the effect of exact acceleration, we collected run-time data for the automata of figure 5.1 on page 65, denoted by P, and figure 5.2 on page 70, denoted by P_A. We used the model checkers UPPAAL and KRONOS – both with a breadth-first search order – to verify whether or not

	P		P_A	
	Mem	Time	Mem	Time
`LARGE`	[kB]	[s]	[kB]	[s]
10^3	1084	0.05	1084	0.01
10^4	1488	2.98	1084	0.01
10^5	6312	374	1084	0.01
10^6	–	†	1084	0.01

Table 5.1: Run-time data obtained with UPPAAL.

	P	P_A
`LARGE`	#	#
10^2	45	21
10^3	432	21
10^4	4290	21
$1,5 \cdot 10^4$	6432	21

Table 5.2: Run-time data obtained with KRONOS.

location `L4` is reachable.Table 5.1 shows the time and memory consumption of UPPAAL as a function of the value of `LARGE`. The † in the table means that we ran out of patience: the model checking takes more than 10 minutes. Similarily, table 5.2 shows the number of states (in the table denoted by #) that KRONOS explored as a function of the value of `LARGE`.

We can explain the constant time and memory consumption and the constant number of explored states of P_A by theorem 5.2. The breadth-first search order ensures that the appended cycle is explored before the complete exploration of the acceleratable cycle. The resulting symbolic state swallows all the smaller symbolic states that result from execution of the acceleratable cycle.

5.2 Acceleration of UPPAAL models

In chapter 4 we proposed a translation from UPPAAL models to RCX byte code and a model of the runtime behavior of that byte code. As an example we applied our compiler to a simple UPPAAL model, see section 4.3 on page 51. The UPPAAL process of the run-time behavior of the byte code, depicted in figure 4.14 on page 56, exhibits many cycles. It may seem that many of these cycles are acceleratable cycles. However, there are differences with the theory developed in the previous section.

First, UPPAAL models can use bounded integer variables. These are not treated in our acceleration theory. Second, UPPAAL models consist of a *network* of timed automata. Third, UPPAAL models can use urgent locations and channels and committed locations.

In section 5.2.1 we explain the consequences of the three differences for our acceleration technique. Fortunately, our technique is not rendered useless, and in 5.2.2 we discuss how we enhanced our compiler `uppaal2rcx` with acceleration. Finally, in section 5.2.3 we show experimental results obtained by application of the compiler to the Level Crossing and Production Cell.

5.2.1 Does equivalence of reachability disappear?

In this section we discuss the consequences of the presence of bounded integer variables, networks of parallel automata, and urgency and commitment for our acceleration technique.

Bounded integer variables

Consider a timed automaton, say P, which uses bounded integer variables as UPPAAL does. Moreover, let us assume that it contains an acceleratable cycle, say A. The edges of A can contain guards over the bounded integer variables, and assignments of the bounded integer variables.

We argue that if A only contains guards over the bounded integer variables and assignments which in total do not change the variable interpretation, then acceleration of P with A still remains exact.

It is clear that the *bounded* integer variables of P can be encoded in the control structure of P, resulting in a new timed automaton without bounded integer variables, say, P_v. Each location of P_v is the product of a variable interpretation and a location of P. Thus, each state (l, ν, v) in the transition system of P has an equivalent state (l', ν) in the transition system of P_v. For example, a state $(l_1, x = 3, a = 5)$ has $((l_1, a = 5), x = 3)$ as equivalent state.

Now let us consider what happens with A in the "variable abstraction" P_v. Let us assume that the edges of A traverse the locations $l_1, .., l_n, l_1$. The acceleratable cycle A will have a number of "equivalent" paths in P_v. Such an equivalent path traverses the locations $(l_1, v_1), .., (l_n, v_n), (l_1, v_{n+1})$. Note that our requirement concerning the integer assignments on A ensures that $v_1 = v_{n+1}$, which means that these equivalent paths are cycles! Moreover, these cycles match A with respect to the clock guards and invariants. Therefore, these cycles are all acceleratable. Note that the number of equivalent cycles depends on the integer guards of A.

If we accelerate A in automaton P, then the appended cycle will also have a number of equivalent paths in P_v. Following a similar argument as above, these paths will be cycles. Moreover, they will exactly be the appended cycles of the equivalent paths of A in P_v. Thus, if we accelerate A in P, then exactly all equivalent paths of A in P_v are also accelerated.

However, if A contains, e.g., only the assignment $n := n + 1$, then the previous informal argument does not hold: the equivalent paths in P_v are not cycles, since the variable interpretation changes when the edges of A are executed.

Networks of timed automata

The following example demonstrates that equivalence of reachability is not retained in a setting of parallel automata that communicate with each other. Consider the UPPAAL model that consists of process P, depicted in figure 5.4, and process *Env*, depicted in figure 5.5. Process P has a local clock `y` and process *Env* has a local clock `z`. The reader can verify that the cycle of process P is an acceleratable cycle, say A. Here we see the first problem:

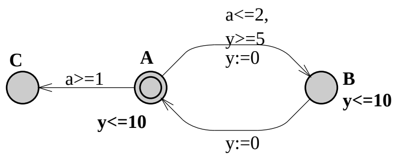


Figure 5.4: Process P.

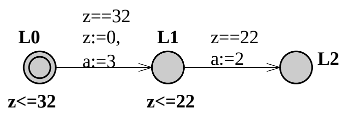


Figure 5.5: Process *Env*.

the acceleratable cycle also contains invariants on clock `z` due to the parallel composition of the automata. However, it can be shown that the acceleration of a cycle as in definition 5.4 which contains additional invariants, remains exact.

The fact that the processes communicate through the shared integer variable `a`, however, poses another problem. Using UPPAAL we verified that the following property is *not* satisfied for $P \,||\, Env$

$$\exists\Diamond(\texttt{P.A} \wedge \texttt{a==2})$$

However, if we add the acceleration to P – we use the model $Acc(P, A) \,||\, Env$ –, then the property *is* satisfied. A similar counter example can be constructed for communication through channels instead of shared integer variables.

The problem originates from the fact that process *Env* can execute an edge while $Acc(P, A)$ is in the appended cycle. If one considers the parallel composition of $Acc(P, A)$ with *Env*, then we see that the appended cycle has been transformed to a structure with more than one "exit". These additional exits correspond to location switches of *Env* while $Acc(P, A)$ is in the appended loop. Clearly, this structure is not an acceleration as in definition 5.5.

We conclude that in a setting of networks of timed automata we have to accelerate the cycles in the parallel composition instead of the cycles in the individual components. For example, figure 5.6 depicts this parallel composition of P with *Env*. The small cycle of process P appears three times in the syntactical parallel composition, since *Env* has three locations. Note that such a parallel composition results in an exponential growth of (i) the size of the system description, and (ii) the number of acceleratable cycles. The size of the reachable symbolic state space, however, will not change.

Due to the exponential growth of the system description, we would rather not construct the parallel composition. By expansion of our technique, we can avoid this. Let us consider the example of $Acc(P, A) \,||\, Env$ again. First, we declare an extra boolean variable `b`, which is initialized to 0. Second, we add two assignments to component $Acc(P, A)$: we add `b:=1` to the first edge of the appended cycle and we add `b:=0` to the last edge of the appended cycle. Third, we guard *all* edges of *Env* with `b==0`. This ensures that *Env*

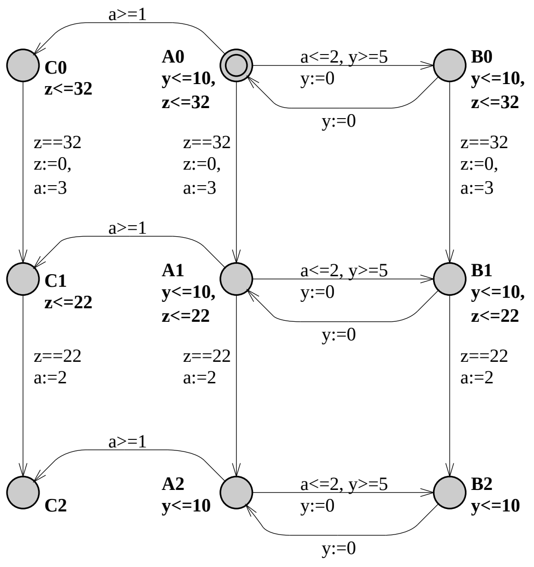


Figure 5.6: The parallel composition of P with *Env*.

can only execute an edge, if $Acc(P, A)$ is not in the appended cycle. We claim that this results in the same effect as acceleration of the three small cycles in figure 5.6. Indeed, when we check the aforementioned reachability property in this adjusted accelerated system, then the property is not satisfied, which is good!

Urgency and commitment

In the previous sections we claimed that we can exactly accelerate single components in a network of timed automata, which use bounded integer variables in a slightly restricted way. This covers a large subset of Uppaal models. The remaining models all use urgent channels or locations, or committed locations.

Let us consider a Uppaal model with three processes, P_1, P_2 and P_3. Process P_1 has an acceleratable cycle starting in location l_1. Process P_2 and P_3 can synchronize over some urgent channel, when they are in locations l_2 respectively l_3. Now consider a situation where the processes are in the mentioned locations. Due to the urgency of the synchronization between P_2 and P_3, no time may elapse as long as it is enabled. We will show that this behavior is not guaranteed when P_1 is accelerated. Therefore, consider the accelerated version of P_1 (including the extra boolean variable that blocks P_2 and P_3 when P_1 is in the appended cycle). Now P_1 can enter its appended cycle, which disables the urgent synchronization. However, time certainly elapses in the appended cycle. It is clear that the urgent synchronization should have been taken before entering the appended cycle. This cannot be expressed with syntactical means.

Concluding, the presence of urgent synchronizations in a model can certainly remove the exactness of our technique. The presence of urgent or committed locations probably does not compromise the exactness. If the exactness is lost, however, then we still have an over approximation of the reachable symbolic state space. Let M be a component of a network of timed automata, let A be an acceleratable cycle of M and let Q be a state property of M, then:

$$M \models \exists\Diamond(Q) \Rightarrow \mathit{Acc}(M, A) \models \exists\Diamond(Q)$$

This is easily proven – since all traces of M are certainly also traces of $\mathit{Acc}(M, A)$ – and does not only hold for our syntactical addition, but for any syntactical addition to the original component. This is the base of Möllers "parking approach" [Möl02]. The consequences for theorem 5.2 are not clear.

When is equivalence of reachability preserved?

In the previous sections we discussed how we can generalize our exact acceleration technique for single timed automata to the more general Uppaal models. Let us consider a single component M of a Uppaal model, which has a cycle A. We *claimed* that we can accelerate this cycle in an exact way, if the following conditions are satisfied:

- if the bounded integer guards and assignments on edges of A are left out, then A is an acceleratable cycle as in definition 5.4, and
- the consecutive execution of the integer assignments on the edges of A does not have a net effect, and
- the total model does not contain urgent synchronizations.

Our generalized exact acceleration technique needs an additional boolean variable, which ensures that no component other than the accelerated component executes an edge, if the accelerated component is in its appended cycle.

When one of the previous conditions does not hold, then our generalized technique is not exact, but an over-approximation. This is still useful to verify the truth of invariance or untruth of reachability properties.

5.2.2 Automatic application of acceleration

In the previous section we discussed the consequences for the exactness of our acceleration technique when it is applied to single components of Uppaal models. We claimed that our technique – extended with the extra boolean variable – remains exact if there is no total effect of the integer assignments on the edges of the acceleratable cycle, and if the concepts of urgency and commitment are not used. Otherwise, our technique still results in an over approximation.

Figure 4.14 on page 56 shows a typical example of a byte code process that results from our translation. Many acceleratable cycles are present in this process. At the end of the previous chapter we explained that such a byte code process displays busy-waiting behavior that fragments the symbolic state space in an unnecessary way. This busy-waiting is in fact the cycling through the various “test and branch” and “branch always far” instructions that implement the control structure. These so-called *idle cycles* thus represent the continuous testing of all guards by the byte code. For example, the cycle `S4`, `S11`, `S12`, `S16`, `S17`, `S21`, `S22`, `S29`, `S33`, `S34`, `S4` is such an idle cycle.

We implemented the acceleration of these idle cycles. Remember that we are able to provide the lower and upper bounds of instructions to the compiler. If we set these bounds wide enough for the “test and branch” and “branch always far” instructions, then many of the idle cycles will satisfy the window precondition of theorem 5.1. Since these cycles only consist of “test and branch” and “branch always far” instructions, there are no integer assignments on the edges of the cycle. Moreover, we implemented the scheme with the extra boolean variable, which assures that other processes cannot execute edges when some process is in an appended cycle.

The cycles that we append to processes differ slightly from those in definition 5.5. Let us consider some acceleratable cycle A with windows $[a, b]$. Let G denote the conjunction of *all* guards on the edges of A. Instead of adding $2 \cdot |A| - 1$ new locations, we add 3 three new locations, see figure 5.7 for the new form of the appended cycle. Location `L` is part of the original

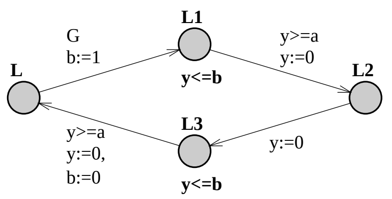


Figure 5.7: The new form of the appended cycle.

automaton, and locations `L1`, `L2`, and `L3` form the appended cycle. The first edge is guarded by G: the conjunction of all guards of the accelerated cycle. Clock y is the clock of the cycle, that implements the delay of the window. On entry of the appended cycle, the extra boolean b is set to 1. On exit, it is reset to 0.

Although we have not (yet!) proven that this new form is equivalent to the old form, it certainly gives an over-approximation. The reason for the introduction of a new form of the appended cycle, is that this new form is much smaller than the original form, which will probably reduce the reachable state space.

Note that a byte code process built from m source processes, each with N_i locations, contains at most $\Pi_{i=1}^{m} N_i$ idle cycles. As we already mentioned, we implemented the acceleration of idle cycles in our compiler `uppaal2rcx`. Using the `-a=<n>` option, the compiler tries to accelerate `n` percent of the maximal number of idle cycles. If cycles are actually accelerated, then the generated model certainly is an over-approximation of the run-time behavior of the generated byte code.

The user can also optionally apply our technique of the blocking boolean variable with the `-g=<b>` option. If `<b>=t`, then the appended cycle is guarded using an extra boolean variable. If `<b>=f`, then the appended cycle is not guarded, with the result that if control of some process is in an appended cycle, then all other processes may execute edges. However, this proves to be very useful when urgent synchronizations are present, as we will see in the next section with experimental results.

Concluding, when we supply the `-a` option to the compiler, and it actually accelerates cycles, then we can say the following about the generated model:

- If we assume that the new form of the appended cycle is equivalent to the old form, and that our assumptions concerning the generalized acceleration technique are correct, then the generated model is exact, if (i) there are no urgent synchronizations in the model, and (ii) the option `-g=t` is used.
- In all other cases – we do not need the assumptions stated above for exact acceleration – the generated model is an over-approximation.

5.2.3 Experimental results

We obtained the following experimental results in almost the same manner as described in section 4.4. The only difference is that we additionally supplied the `-a=100` and `-g=f` options to the compiler. Thus, the compiler tries to accelerate as much idle cycles as possible, and it does not use the extra boolean to guard the appended cycle.

The generated models are over-approximations. As a result, we can only obtain conclusive answers about the truth of invariance and the untruth of reachability properties.

Note that we omitted properties (3.3) and (3.13), which both are concerned with deadlock. These properties will certainly not be satisfied, since the appended cycles introduce deadlock.

Table 5.3 shows us that we can obtain one conclusive answer. It is true that the physical gate cannot reach its error location. This is not unexpected, since the physical gate is solely controlled by the gate controller, which is the same as in the conceptual design; only the train is implemented. Comparison with table 4.1 on page 58 shows us that the accelerated model is much faster than the convex hull approximation.

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(3.4)	2	3	`-A`	maybe
	1 : 22	21		maybe
(3.5)	< 1	2		maybe
(3.6)	1 : 45	8	`-A`	**YES**

Table 5.3: Accelerating the implementation of the train controller.

Table 5.4 contains the data concerning the implementation of the gate controller. Unfortunately, we were not able to obtain conclusive answers.

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(3.4)	< 1	2		maybe
(3.5)	< 1	2		maybe
(3.6)	3 : 46	3	`-A`	maybe
	4 : 53 : 11	95		?

Table 5.4: Accelerating the implementation of the gate controller.

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(4.1)	24	3	`-A`	maybe
	4 : 46 : 16	91		?
(3.10)	< 1	2		maybe

Table 5.5: Accelerating the implementation of the belt controller.

The results in table 5.5 show that model checking property (4.1) with use of the convex hull approximation takes more time then when no acceleration is applied. This can be explained by the contribution of the over-approximating acceleration to the reachable state space.

Fortunately, table 5.6 shows us some good news: we are able to obtain two conclusive answers within reasonable time. Property (3.9) specifies the upper bound of 48 seconds on the time that it takes to transport a brick to its destination. Property (3.10) specifies that the brick does not reach

Property	Time [h:m:s]	Mem [MB]	Options	Satisfied?
(3.8)	32 : 50	142	`-A`	maybe
(3.9)	51 : 53	181	`-A`	**YES**
(3.10)	40 : 20	181	`-A`	**YES**

Table 5.6: Accelerating the implementation of the robot arm controller.

its error location. At least this gives us some faith in the correctness of the implementation of the robot arm controller. Comparison with table 4.4 on page 60 shows us that applying the acceleration results in a speed-up of the model checking process with *at least* a factor 5.

It is noteworthy that we also did the previous model checking runs with models which we generated using the `-a=100` and `-g=t` option. Thus, these models use our technique to "atomize" the execution of appended cycles. We could not obtain any conclusive answers using these models. Analysis of counter example traces provided by UPPAAL confirmed what we already suspected. The extensive use of urgent synchronizations in our models, e.g. the `hurry` channel, led to the behavior we described in section 5.2.1 on page 75.

5.3 Summary

In this chapter we have presented an acceleration technique for forward symbolic reachability analysis of timed automata. Our technique addresses the fragmentation problem that occurs when different time scales are present in a timed automaton. Our technique is applicable to a subset of timed automata, namely those that contain *acceleratable* cycles. We append an extra cycle to the timed automaton that in *one* execution computes the result of the *iterated* execution of the acceleratable cycle in the original automaton. Whether or not a cycle is acceleratable, and the form of the appended cycle are easily computable from the syntax of the timed automaton.

We have proven that our syntactic adjustment is exact with respect to reachability properties and that it will speed up forward symbolic reachability analysis with a breadth-first search order. Using the model checkers UPPAAL and KRONOS, we have demonstrated that our technique can seriously reduce the time and memory consumption of the model check process. It even completely solves the fragmentation problem for our theoretical example.

We generalized our technique to general UPPAAL models and argued that it remains exact if (i) the integer assignments of an acceleratable cycle have no net effect, and (ii) there are no urgent synchronizations in the model. Moreover, we automatically applied our technique to *idle cycles* in UPPAAL models of the run-time behaviour of executable byte code for the Level Crossing and Production Cell. We were able to obtain three conclusive answers to invariance properties within reasonable time. Remember that we were not able to obtain a single conclusive answer without acceleration!

Future work. A logical next step is to prove our claims concerning the generalization of our acceleration technique to UPPAAL. Especially, we would like to investigate the precise effect of urgent channels on our technique. Their semantics are very effective for modeling reactive systems, but they render our technique inexact. However, we think that it is very useful to achieve exact acceleration, since that enables us to verify the truth of reachablity properties. As we already sketched, urgent synchronizations should be executed before entering an appended cycle. Thus, there is need for more sophisticated guiding; not only just breadth-first search order. The recent advancements in the area of *guided model checking* might be helpful [HLP00, BFH$^+$01].

Another interesting subject is investigation of weakening of the constraints on acceleratable cycles, as used in this chapter. We can probably permit upper bounds on the clock of the cycle and lower bounds on the other clocks on the edges of the cycle.

Finally, we note that a compressed version of this chapter has been accepted at the Workshop on Theory and Practice of Timed Systems 2002 [HL02].

Chapter 6

Conclusion

In this master's thesis we introduced a technique for the development of reactive programs using a model checker. In short, our technique consists of a translation from a conceptual design to (i) executable code that (approximately) implements this design, and (ii) a model of the run-time behavior of the generated code. See figure 1.1 on page 3 for a graphical depiction. The conceptual design is used to validate the design, and the generated models are used to verify the implementation of the design.

In order to answer the two main questions concerning our technique, we implemented it for a fixed hardware platform and model checker, which we have chosen in chapter 2. We use the LEGO RCX micro controller for its availability, flexibility and realism. As a result, the generated executable code is so-called byte code, an assembly like language that is interpreted by the firmware of the RCX. We use the formalism of timed automata, in the shape of the model checker UPPAAL, for the conceptual modeling and for the modeling of the run-time behavior of the system.

Of course, an implementation without input is of no use. In chapter 3 we constructed conceptual designs of existing LEGO setups – the Level Crossing and the Production Cell – both controlled by two RCX bricks. The simulator and the option to generate traces of a counter example that disproves a specification property, were very useful during the construction of the conceptual designs. They allowed us to quickly track and solve logical design errors.

In chapter 4 we described and implemented the translation, which is the heart of our technique to obtain correct reactive programs for the RCX using UPPAAL. We explained that a bare UPPAAL model does not contain enough information to implement it on an arbitrary hardware platform. Therefore, the user must supply information about the intention of the various processes, variables and clocks in the model.

We also explained that the implementation of the conceptual design will – in general – differ from the conceptual design due to the fact that UPPAAL supports unrealizable (for the RCX at least) concepts like concurrency and non-determinism. In order to avoid a too large gap between the conceptual design and its implementation, we stated some requirements on the conceptual design. The remaining small gap is covered by the generated model of the run-time behavior of the implementation.

Finally, we implemented the translation and compiled the Level Crossing and the Production Cell. The generated code for the Level Crossing functions as expected. The code for the Production Cell, however, only functions partially. Due to lack of time we did not fix this. Verification of the generated byte code proved to be practically impossible, which is mainly caused by the so-called *fragmentation problem*, which occurs when different time scales are used in models.

In chapter 5 we developed theory which addresses the fragmentation problem. We propose a syntactic adjustment to a subset of timed automata which is exact with respect to reachability properties and which speeds up forward symbolic reachability analysis, if a breadth-first search order is used. We have demonstrated that our technique completely solves the fragmentation problem for a theoretical example.

Unfortunately, the Uppaal models which are generated by our compiler are – in general – not contained in the class of acceleratable timed automata. Therefore, we generalized our technique to Uppaal models, but it seems that the use of urgent channels disturbs the exactness. However, our generalized technique always remains an over-approximation.

Finally, we enhanced our compiler with the generalized acceleration technique. Our compiler can thus generate two kinds of models of the run-time behavior, which we already planned in figure 1.2 on page 4. We applied the acceleration to the Level Crossing and the Production Cell. The results are promising: we can actually verify three invariance properties within reasonable time. A compressed version of this chapter has been accepted by the workshop TPTS02 [HL02].

We now can answer the first research question of this thesis, which asked: "What are the benefits of a translation from models appearing in the conceptual design phase to (i) executable code and (ii) models of that code, for the development of reactive programs for embedded real-time systems?" We see two obvious benefits of such a translation.

First, automated code generation saves much time and it may prevent some errors, e.g., errors due to concentration loss of the programmer. This is clearly illustrated by the generated byte code for the Level Crossing and the Production Cell, which is fairly complicated. Moreover, we found that – in this particular setting – the translation does not need to restrict the models of the conceptual design phase in some fundamental way. E.g., all features of Uppaal can be used.

Second, the relation between the conceptual design and the implementation has been formalized by the translation. If the compiler ensures that the (mostly unpreventable) gap between a conceptual design and its implementation is small, then validation of the conceptual design is meaningful. In other words, many logical design errors can be detected during the conceptual design phase. In chapter 4 we have explained that – in this particular setting – a compiler can actually approximate one execution of the conceptual design. The fact that the generated code for the Level Crossing worked immediately illustrates this.

The benefits of models of the run-time behavior of generated byte code are not clear, since they are practically unverifiable. This brings us to the second question of this thesis , which asks “How can we accelerate the model checking process of the model of the executable code?”

We have shown that we can (partially) handle the fragmentation problem, which is one of the main reasons that inhibit the practical verification of byte code, in the setting of a single timed automaton without bounded integer variables. We argued that generalization of this *exact acceleration* technique to Uppaal is probably not too difficult and can be achieved by formalization and proof of our thoughts in section 5.2.1. However, the role of urgent synchronizations is not yet clear. Since they are very useful for modeling conceptual reactive systems, we would rather not exclude them.

Concluding, we think that the integration of model checkers in the development process of reactive programs can be very valuable. Especially in a setting where timing parameters may have very subtle and unexpected effects, model checkers can provide a relative easy way to detect errors in conceptual designs. Moreover, in many cases it will be possible to define a sensible translation from models which appear in the conceptual design phase to executable code. This translation saves implementation time and reduces the risk of errors. Verifying executable code using a model checker still is very difficult. However, we think that our research concerning the fragmentation problem brings this yet a little closer.

References

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized System Verification. In *11th International Conference on Computer Aided Verification*, number 1633 in LNCS, pages 134–145. Springer–Verlag, 1999.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993.

[AD90] R. Alur and D.L. Dill. Automata for modeling real-time systems. In *17th International Colloquium on Automata, Languages, and Programming*, pages 322–335, 1990.

[Alu99] R. Alur. Timed Automata. In *11th International Conference on Computer Aided Verification*, number 1633 in LNCS, pages 8–22. Springer–Verlag, 1999.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BFH$^+$01] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *8th International Conference on Computer Aided Verification*, number 1102 in LNCS, pages 1–12. Springer–Verlag, 1996.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *4th International Static Analysis Symposium*, LNCS. Springer–Verlag, 1997.

[BH97] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with non-regular sets of configurations. In *24th International Colloquium on Automata, Languages, and Programming*, number 1256 in LNCS. Springer–Verlag, 1997.

[Boe79] B.W. Boehm. Software engineering: R & D trends and defense needs. In *Research Directions in Software Technology*, 1979.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *6th International Conference on Computer Aided Verification*, number 808 in LNCS, pages 55–67. Springer–Verlag, 1994.

[CDH$^+$00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000. Available through URL `http://www.cis.ksu.edu/santos/bandera/`.

[CK96] E. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[CL00] F. Cassez and K. G. Larsen. The Impressive Power of Stopwatches. In *International Conference on Concurrency Theory*, pages 138–152, 2000.

[Die99] H. Dierks. *Specification and Verification of Polling Real-Time Systems.* PhD thesis, Carl von Ossietzky Universität Oldenburg, July 1999.

[Dil89] D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 197–212. Springer–Verlag, 1989.

[Fag86] M.E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

[Hen96] T. A. Henzinger. The Theory of Hybrid Automata. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.

[Hen01] M. Hendriks. Translating UPPAAL to Not Quite C. Technical Report 8, CSI University of Nijmegen, March 2001.

[HHWT97] T. A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[HKPV98] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[HL02] M. Hendriks and K. G. Larsen. Exact Acceleration of Real-Time Model Checking. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, April 2002.

[HLP00] T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs Using UPPAAL. In Ten H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22. IEEE Computer Society Press, April 2000.

[Hun00] T. S. Hune. Modeling a language for embedded systems in timed automata. Technical Report RS-00-17, BRICS, Basic Research in computer Science, August 2000. 26 pp. Earlier version entitled *Modelling a Real-Time Language* appeared in FMICS99, pages 259–282.

[IKL$^+$00] T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen. Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using UPPAAL. In *IEEE Euromicro Conference on Real-Time Systems*, pages 147–155, 2000.

[LPY] M. Lindahl, P. Pettersson, and W. Yi. Formal Design and Analysis of a Gear-Box Controller.

[LPY98] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. In *International Journal on Software Tools for Technology Transfer*, 1, pages 134–152. Springer-Verlag, 1998.

[LRRA98] P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann. Qualitätssicherung Software-basierter technischer Systeme – Problembereiche un Lösungsansätze. *Informatik Spektrum*, 21:249–258, 1998.

[Möl02] M. O. Möller. Parking Can Get You There Faster. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, April 2002.

[Nie00] S. Nielsson. Introduction to the legOS kernel, 2000. Available through URL `http://legos.sourceforge.net/`.

[Pro99] K. Proudfoot. RCX Internals, 1998–1999. Available through URL `http://graphics.stanford.edu/~kekoa/rcx/`.

[PS00] A. Pnueli and E. Shahar. Liveness and Acceleration in Parameterized Verification. In *12th International Conference on Computer Aided Verification*, number 1855 in LNCS, pages 328–343. Springer–Verlag, 2000.

[Som96] I. Sommerville. *Software engineering.* Addison–Wesley, 1996.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice–Hall, 1996.

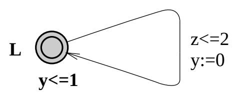
[Yov97] S. Yovine. KRONOS: a verification tool for real-time systems. In *Software Tools for Technology Transfer*, 1997.

Appendix A

Proofs of lemmas and theorems

Lemma 5.1 *Not every cycle has a window.*

Proof. Consider the timed automaton M with two clocks, y and z, depicted below. Obviously, there is a cycle, but this cycle does not have a window.



We prove this by contradiction. Let us assume that the cycle does have a window. Now consider the following trace of this timed automaton (we denote the clock interpretation ν as a binary tuple that first contains the value $\nu(y)$ and second the value $\nu(z)$):

$$((\mathtt{L},(0,0)),(\mathtt{L},(1,1)),(\mathtt{L},(0,1)),(\mathtt{L},(1,2)),(\mathtt{L},(0,2)),(\mathtt{L},(0,2)),(\mathtt{L},(0,2)))$$

Observe that this is a compressed trace in $Tr(M)$ that executes the cycle three times. From the invariant of L can be concluded that the cycle execution takes at most 1 time unit. Moreover, the first execution can take no time, since the cycle is not guarded by a lower bound on some clock. Therefore, we must conclude that the window of this cycle is $[0,1]$.

If we consider the third execution of the cycle in the trace above, we see that the first bullet of definition 5.3 is satisfied. However, the second bullet is not satisfied, since we cannot delay more than zero time units because then the value of z would exceed 2, which disables the cycle. Thus, we must conclude that the window is $[0,0]$.

This window of $[0,0]$ is not valid for the first execution of the cycle in the trace above by the first bullet of definition 5.3. Thus, we conclude that this cycle has no window.

□

Lemma 5.2 (Cycle consecution) *Let (E_c, y) be an acceleratable cycle of some timed automaton M. If E_c is 1-time executable, then it is m-times* *executable, for all $m > 0$.*

Proof. Let us assume that E_c is 1-time executable. From definition 5.2 we know that a finite compressed trace in $Tr(M)$ exists with a suffix as in definition 5.2, say of the form

$$((l_0, \nu_0), (l_0, \nu_0'), (l_1, \nu_1), ..., (l_{k-1}, \nu_{k-1}'), (l_k, \nu_k))$$

By definitions 5.1 and 5.2, we can conclude that this suffix starts and ends in location $src(e_0)$, in other words $l_0 = l_k = src(e_0)$. Moreover, since we consider an acceleratable cycle, we know that $\nu_k(y) = 0$. We can extend this suffix with a δ-delay transition, where $\delta = \nu_0'(y)$, and we obtain the suffix

$$((l_0, \nu_0), (l_0, \nu_0'), (l_1, \nu_1), ..., (l_{k-1}, \nu_{k-1}'), (l_k, \nu_k), (l_k, \nu_k'))$$

where $\nu_k'(y) = \nu_0'(y)$. This delay transition is allowed because $l_k = l_0$, and $\nu_k'(y) = \nu_0'(y)$. Since we consider an acceleratable cycle, we know that the invariant of l_k is only concerned with clock y. Therefore, we can conclude that $\nu_k' \models I(l_k)$.

As a result, we can repeat the sequence of action and delay transitions of the first part of the trace an arbitrary number of times, since the guards and invariants are solely concerned with clock y.

□

Lemma 5.3 (Window computation) *Each acceleratable cycle has a window.*

Proof. Consider a timed automaton $M = (L, l^0, \Sigma, X, I, E)$ and an acceleratable cycle $((e_0, e_1, ..., e_{n-1}), y)$. We will show that we can effectively compute the window from the syntax of the timed automaton.

Let the tuple $(l_i, a_i, \phi_i, \lambda_i, l_{i+1})$ denote edge e_i. We can find p natural numbers $0 \leq k_0 < k_1 < ... < k_{p-1}$ that exactly correspond to the indices of the edges on which clock y is reset. Since we know by definition that y is reset on edge e_{n-1}, p is at least one. Next, we compute the following numbers for $0 \leq j < p$ (we define $k_{-1} = -1$):

$$\begin{array}{rcl} a_{k_j} & = & max\,\{\, cn_g(\phi_i) \,|\, k_{j-1} < i \leq k_j \,\} \\ b_{k_j} & = & cn_I(I(l_{k_j})) \end{array}$$

Since we consider the guards and invariants of an acceleratable cycle, all the numbers a_{k_j} and b_{k_j} are defined. We will show that the acceleratable cycle has a window of

$$\left[\sum_{j=0}^{p-1} a_{k_j} \,,\, \sum_{j=0}^{p-1} b_{k_j} \right]$$

Let us consider a subsequence of some compressed trace as in definition 5.3. This subsequence denotes one execution of the acceleratable cycle, starting in the reset location. We can partition this subsequence by the numbers k_j, to obtain p "partial" compressed subsequences:

$$\begin{array}{ll}
((l_0,\nu_0),...,(l_{k_0},\nu'_{k_0}),(l_{k_0+1},\nu_{k_0+1})) & \text{part } k_0 \\
((l_{k_0+1},\nu_{k_0+1}),...,(l_{k_1},\nu'_{k_1}),(l_{k_1+1},\nu_{k_1+1})) & \text{part } k_1 \\
\vdots & \\
((l_{k_{j-1}+1},\nu_{k_{j-1}+1}),...,(l_{k_j},\nu'_{k_j}),(l_{k_j+1},\nu_{k_j+1})) & \text{part } k_j \\
\vdots & \\
((l_{k_{p-2}+1},\nu_{k_{p-2}+1}),...,(l_{k_{p-1}},\nu'_{k_{p-1}}),(l_0,\nu'_0)) & \text{part } k_{p-1}
\end{array}$$

For each of these parts holds that clock y is reset with the last action transition. Moreover, the guards and invariants in these parts satisfy the special forms defined in definition 5.4.

We will show that (i) the total delay in a part k_j, i.e., $\nu'_{k_j}(y)$, is an element of $[a_{k_j}, b_{k_j}]$ and (ii) that we can adjust the amount of delay in part k_j to d for all $d \in [a_{k_j}, b_{k_j}]$, such that the adjusted sequence is a subsequence of some trace in $Tr(M)$.

Part (i). Let us consider part k_j. This fragment traverses part of the cycle until y is reset. We know by computation of the numbers k_j that $\nu_{k_{j-1}+1}(y) = 0$. This also holds for k_0, since $l_0 = src(e_0)$ and clock y is reset on every incoming edge of the reset location, or $\nu_0(y) = 0$ by definition of ν_{init}.

We show that $\nu'_{k_j}(y) \in [a_{k_j}, b_{k_j}]$. Since the part is from an existing trace, the clock interpretation in part k_j must have satisfied the maximal guard $y \geq a_{k_j}$ at some moment. The clock y is only reset on the last transition in the part and therefore $\nu'_{k_j}(y) \geq a_{k_j}$. We also know that $\nu'_{k_j} \models I(l_{k_j})$ and thus that $\nu'_{k_j}(y) \leq cn_I(I(l_{k_j})) = b_{k_j}$. As a result, the total delay of this part is an element of $[a_{k_j}, b_{k_j}]$.

Part (ii). Consider the part k_j, for simplicity denoted by

$$((l_q,\nu_q),(l_q,\nu'_q),(l_{q+1},\nu_{q+1}),...,(l_{r-1},\nu_{r-1}),(l_{r-1},\nu'_{r-1}),(l_r,\nu_r)) \qquad (6.1)$$

We will show that, given a $d \in [a_{k_j}, b_{k_j}]$, we can redefine the delays in each location such that the total delay of the part equals d. In other words, we construct a new finite compressed trace of the form

$$((l_q,\mu_q),(l_q,\mu'_q),(l_{q+1},\mu_{q+1}),...,(l_{r-1},\mu_{r-1}),(l_{r-1},\mu'_{r-1}),(l_r,\mu_r)) \qquad (6.2)$$

where that $\mu_q = \nu_q$ and the total delay in this subsequence equals d. More formally, $\mu_r(x) = \nu_r(x) + d$ for all clocks $x \neq y$, and of course $\mu_r(y) = \nu_r(y)$, since y is reset on the last edge. We inductively define the clock interpretations μ as follows for every $q \leq i < r-1$:

$$\begin{aligned}
&\mu_q = \nu_q \\
&\mu'_i = \mu_i + \delta_i \\
&\mu_{i+1} = \mu'_i \\
\\
&\delta_i = 0 && \text{iff } \mu_i \models \phi_i \\
&\delta_i = cn_g(\phi_i) - \mu_i(y) && \text{otherwise}
\end{aligned}$$

These definitions mean that if the next action transition is already enabled, then the delay in the location is zero. If not, the delay is the minimal delay needed to enable the edge. We thus construct trace 6.2 up to the state (l_{r-1}, μ_{r-1}). We choose the last delay after we have proven that a valid compressed trace results from our construction. We prove by induction on i the following for all $q \leq i < r-1$:

(1) Enabledness of edges: $\mu_i \models I(l_i)$, $\mu'_i \models I(l_i)$, $\mu'_i \models \phi_i$, and $\mu'_i \models I(l_{i+1})$.

(2) We construct a “faster” trace then we have: $\mu'_i(y) \leq \nu'_i(y)$.

(3) We construct the “fastest” trace: $\mu'_i(y) \leq a_{k_j}$.

Base. We know that $\mu_q(y) = 0$ and by the existence of trace (6.1) that $\mu_q \models I(l_q)$. By choice of δ_q, we know that $\mu'_q(y) = cn_g(\phi_q)$, and thus $\mu'_q \models \phi_q$. Now we must show that $\mu'_q \models I(l_q)$ and $\mu'_q \models I(l_{q+1})$. Consider the state (l_q, ν'_q) in trace 6.1. It is clear that ν'_q satisfies the guard and therefore $\nu'_q(y) \geq \mu'_q(y)$. Moreover, ν'_q satisfies the invariants of locations l_q and l_{q+1}. Since $\nu'_q(y) \geq \mu'_q(y)$, we can conclude that $\mu'_q(y)$ must also satisfy $I(l_q)$ and $I(l_{q+1})$. This proves the properties 1 and 2.

We know by definition that $\mu_q(y) = 0$. So if $\mu_q \models \phi_q$, then $\mu'_q(y) = 0$ and thus $\mu'_q(y) \leq a_{k_j}$, since a_{k_j} is at least 0. On the other hand, if $\mu_q \not\models \phi_q$, then $\mu'_q(y) = \delta_q = cn_g(\phi_q)$. In this case, we know by definition that a_{k_j} is at least $cn_g(\phi_q)$, and thus $\mu'_q(y) \leq a_{k_j}$. This proves property 3 and the base of the induction.

Induction. Suppose that the properties 1, 2 and 3 hold for $i = m$, where $q \leq m < r-2$. We will show that they also hold for $i = m+1$.

Since no clocks are reset on edge e_m, it is true that $\mu_{m+1} = \mu'_m$. From part 1 of the induction hypotheses follows that $\mu_{m+1} \models I(l_{m+1})$. By choice of δ_{m+1}, we know that $\mu'_{m+1} \models \phi_{m+1}$. Next, consider the state (l_{m+1}, ν'_{m+1}) in trace (6.1). We show that $\mu'_{m+1}(y) \leq \nu'_{m+1}(y)$. There are two cases:

- Suppose $\delta_{m+1} = 0$, then $\mu'_{m+1} = \mu_{m+1}$. Since $\mu'_{m+1} = \mu_{m+1} = \mu'_m$, we obtain by the induction hypothesis that $\mu'_{m+1}(y) \leq \nu'_m(y)$. We know by definition that no clocks are reset on this edge and thus $\nu'_m(y) \leq \nu'_{m+1}(y)$. Combining these last two formulas gives us that $\mu'_{m+1}(y) \leq \nu'_{m+1}(y)$.

- Suppose $\delta_{m+1} > 0$. Using the minimal choice of δ_{m+1} we can conclude that $\mu'_{m+1}(y) \leq \nu'_{m+1}(y)$. For a proof, suppose that $\mu'_{m+1}(y) > \nu'_{m+1}(y)$. Filling in the definition of μ'_{m+1} and δ_{m+1} gives us that $\mu'_{m+1}(y) = cn_g(\phi_{m+1})$. Combination of this formula with our assumption results in $\nu'_{m+1}(y) < cn_g(\phi_{m+1})$. In other words, the guard ϕ_{m+1} in trace (6.1) is not enabled from state (l_{m+1}, ν'_{m+1}). This clearly contradicts the existence of trace (6.1) and we conclude that $\mu'_{m+1}(y) \leq \nu'_{m+1}(y)$.

Next, we show that $\mu'_{m+1} \models I(l_{m+1})$ and that $\mu'_{m+1} \models I(l_{m+2})$. Above, we already proved that property 2, $\mu'_{m+1}(y) \leq \nu'_{m+1}(y)$, holds for the induction step. From the existence of trace (6.1), it is clear that $\nu'_{m+1} \models I(l_{m+1})$ and $\nu'_{m+1} \models I(l_{m+2})$. From the special form of the invariants, they give upper bounds on clock y, we can conclude that $\mu'_{m+1} \models I(l_{m+1})$ and $\mu'_{m+1} \models I(l_{m+2})$. This proves the properties 1 and 2.

Now only property 3 is left. We show that $\mu'_{m+1}(y) \leq a_{k_j}$. Since no clocks are reset on edge e_m, we know that $\mu'_m = \mu_{m+1}$. Using part 3 of the induction hypothesis, we obtain $\mu_{m+1}(y) \leq a_{k_j}$. Next, we distinguish two cases for the delay in location l_{m+1}:

- If $\delta_{m+1} = 0$, then $\mu'_{m+1} = \mu_{m+1}$. Combination with the formula above thus easily gives us that $\mu'_{m+1}(y) \leq a_{k_j}$.
- If $\delta_{m+1} > 0$, then we know by the minimal choice of δ_{m+1} that $\mu'_{m+1}(y) = cn_g(\phi_{m+1})$. Clearly, $a_{k_j} \geq cn_g(\phi_{m+1})$ and thereforewe can conclude that $\mu'_{m+1}(y) \leq a_{k_j}$.

Now that we have proven that our choice of delays up to the location l_{r-2} enables all edges, we must define the last delay δ_{r-1}, prove that edge e_{r-1} is still enabled after that delay, and show that the total delay of our compressed fragment equals d.

Since no clocks are reset on edge e_{r-2}, we know that $\mu_{r-1} = \mu'_{r-2}$. By property 1, we conclude that $\mu_{r-1} \models I(l_{r-1})$, and thus $\mu_{r-1}(y) \leq b_{kj}$. By property 3 we know that $\mu_{r-1}(y) \leq a_{k_j}$. Now we can define the last delay as $\delta_{r-1} = d - \mu_{r-1}(y)$. After this delay, the guard on edge e_{r-1} and the invariant of location l_{r-1} are satisfied, since $(\mu_{r-1} + \delta_{r-1})(y) = d$ and $a_{r-1} \leq d \leq b_{r-1}$. The invariant of location l_r is also satisfied because clock y is reset to 0 on edge e_{r-1}. This proves part (ii).

Concluding, the total delay for the concatenation of all partial trace fragments is our window and its bounds are given by the interval. Moreover, using the technique demonstrated in part (ii), we can adjust the delays of the separate parts of the subsequence to obtain any delay that lies within this window. Of course, these separate parts can still be "concatenated" after the adjustment, since the "overlapping" states of the different parts agree on the value of clock y. This enables the inductive adjustment of the delays in the total subsequence.

□

We need the following three small lemmas in our proof of theorem 5.1.

Lemma 6.1 *For all $a, b, m \in \mathbb{N}$, if $3a \leq 2b$ and $m \geq 2$, then $(m+1)a \leq mb$.*

Proof. We prove this by induction on m. The base, $m = 2$, follows directly from our assumption. Now suppose that it holds for $m = n$ and let us consider the case $m = n + 1$:

$$((n+1)+1)a \leq (n+1)b \quad \Leftrightarrow \quad (n+1)a + a \leq nb + b$$

From the induction hypothesis we know that $(n+1)a \leq nb$ and since $3a \leq 2b$ we also know that $a \leq b$. This proves the lemma.

□

Lemma 6.2 *For all $D \in \mathbb{R}^+$ and $a, b \in \mathbb{N}$, if $D \geq 2a$ and $3a \leq 2b$, then there exists a $k \in \mathbb{N}$ such that $D \in [k \cdot a, k \cdot b]$.*

Proof. We prove that $\cup_{n=2}^{k}[na, nb] = [2a, kb]$ by induction on k. The base, $k = 2$, is straightforward to verify. Now let us assume that it holds for $k = m$. The case for $k = m + 1$ looks like this:

$$\begin{aligned} \bigcup_{n=2}^{m+1}[na, nb] &= \bigcup_{n=2}^{m}[na, nb] \cup [(m+1)a, (m+1)b] \\ &= [2a, mb] \cup [(m+1)a, (m+1)b] \end{aligned}$$

From lemma 6.1 we know that $(m+1)a \leq mb$, since $m \geq 2$ and therefore

$$[2a, mb] \cup [(m+1)a, (m+1)b] = [2a, (m+1)b]$$

Since b is strictly greater than a the limit of this union covers all real numbers larger than or equal to $2a$. This proves the lemma.

□

Lemma 6.3 *For all $D \in \mathbb{R}$ and $a, b \in \mathbb{N}$, if there exists a $k \in \mathbb{N}$ such that $D \in [ka, kb]$, then we can write D as $d_1 + d_2 + ... + d_k$, where every d_i is an element of $[a, b]$.*

Proof. We prove this by induction on k. It is easy to see that this holds for $k = 1$, since then $d_1 = D$. Now let us assume that it holds for $k = p$. Then it also holds for $k = p + 1$. Suppose that $D \in [(p+1)a, (p+1)b]$. Then $D = \delta + \Delta$ for some $\delta \in [a, b]$ and $\Delta \in [pa, pb]$. From the induction hypothesis we know that Δ can be written as $d_1 + ... + d_p$, where every d_i is an element of $[a, b]$. The fact that δ also is an element of $[a, b]$ proves that we can write D as $d_1 + ... + d_{p+1}$, where every d_i is an element of $[a, b]$.

□

Theorem 5.1 (Equivalence of reachability) *Let $(L, l^0, \Sigma, X, I, E)$ be a timed automaton M, let A be an acceleratable cycle of M with a window of $[a, b]$, and let ϕ be a reachability properties of M.*

$$3a \leq 2b \Rightarrow (M \models \phi \Leftrightarrow Acc(M, A) \models \phi)$$

Proof. To prove the theorem, assume $3a \leq 2b$.

($\Rightarrow$) Let us assume that $M \models \phi$. Then there exists a trace in $Tr(M)$ that satisfies ϕ. Observe that $Acc(M, A)$ only *extends* M with some locations and edges. Therefore, a trace in $Tr(M)$ is also a trace in $Tr(Acc(M, A))$ and we can conclude that $Acc(M, A)$ also satisfies ϕ.

($\Leftarrow$) Now, let us assume that $Acc(M, A) \models \phi$. We will show that any state (l, ν), where l is a location of M, that is reachable in $Acc(M, A)$, is also reachable in M. We assume the same naming conventions of the locations as in definition 5.5.

Let us consider a finite subsequence of a trace in $Tr(Acc(M, A))$ that starts on entry of $src(e_0)$ and takes the added cycle exactly once:

$$((l_0, \nu_0), ..., (l_0, \nu_i), (l'_1, \nu_{i+1}), ..., (l''_{n-1}, \nu_{f-1}), (l_0, \nu_f))$$

We let D denote the sum of all delay transitions in this subsequence. Note that $D \geq 2a$. Since y is reset on the edge to l_0 (see definition 5.4), we know that $\nu_f(y) = 0$. Moreover, no other clocks are reset on edges of the acceleratable cycle. Therefore, $\nu_f(x) = \nu_0(x) + D$ for all clocks x, except y.

We will show that there exists a $k \in \mathbb{N}$ such that the delay D can be gathered by k executions of the acceleratable cycle in M. In other words, we will show that a finite compressed trace in $Tr(M)$ exists, where $\mu_{0,0}$ equals ν_0, say of the following form:

$$((l_0, \mu_{0,0}), (l_0, \mu'_{0,0}), (l_1, \mu_{1,0}), ..., (l_0, \mu_{0,1}), (l_0, \mu'_{0,1}), ...,$$
$$(l_{n-1}, \mu'_{n-1,k-1}), (l_0, \mu_{0,k}))$$

such that $\mu_{0,k}$ equals ν_f. It is clear that if this trace exists, then the added cycle of $Acc(M, A)$ does not render states reachable that are not reachable in M.

First we observe that the acceleratable cycle in M is 1-time executable, since we assumed that a trace through $Acc(M, E_c)$ exists that takes the appended cycle once. This trace can easily be compressed to obtain the delay in each location of the appended cycle. The appended cycle mimics two unfoldings of the acceleratable cycle, and therefore we can conclude that we can apply the same delay transitions to obtain a compressed trace in $Tr(M)$ that proves that E_c is 1-time executable. By lemma 5.2 we know that we can execute the acceleratable cycle an arbitrary number of times.

Next, by lemma 5.3 we can pick an element of $[a, b]$ to be the delay for each execution of the acceleratable cycle. Since $D \geq 2a$ and we assumed that $3a \leq 2b$, we can apply lemma 6.2 to conclude that there exists a $k \in \mathbb{N}$ such that $D \in [ka, kb]$. With this knowledge we can apply lemma 6.3 to conclude that we can write D as $d_1 + d_2 + ... + d_k$, where every d_i is an element of $[a, b]$.

Concluding, we know that the compressed trace in $Tr(M)$ exists that executes the acceleratable cycle k times from the state $(l_0, \mu_{0,0})$. Moreover, the last clock interpretation in this trace, $\mu_{0,k}$, assigns 0 to clock y, since y is – by definition – reset on all ingoing edges of l_0. The sum of all delay transitions equals D, and therefore $\mu_{0,k}(x) = \mu_{0,0}(x) + D$ for all clocks $x \neq y$. Thus, $\mu_{0,k}$ equals ν_f.

□

Theorem 5.2 (Effectiveness of acceleration) *Let the timed automaton M be defined by $(L, l^0, \Sigma, X, I, E)$ and let $A = ((e_0, ..., e_{n-1}), y)$ be an acceleratable cycle. If y is reset on edge e_0, then all states reachable by more than one execution of the acceleratable cycle in M, are reachable by exactly one execution of the appended cycle in $Acc(M, A)$.*

Proof. We let $(l_i, a_i, \phi_i, \lambda_i, l_{i+1})$ denote edge e_i and we let $[a, b]$ denote the window of A. We assume the same naming conventions as in definition 5.5.

Let us consider a subsequence of some trace of M that starts on entry of the reset location of A and that takes the acceleratable cycle $m > 1$ times. We denote this trace by

$$((l_0, \nu_0), ..., (l_0, \nu_j), ..., (l_{n-1}, \nu_p), (l_0, \nu_q))$$

We know from the properties of acceleratable cycles that $\nu_0(y) = \nu_q(y) = 0$ and $\nu_q(x) = \nu_0(x) + D$ for the other clocks x, where $D \geq 2a$. We show that a compressed trace of $Acc(M, A)$ exists with the following subsequence

$$((l_0, \nu_{0,0}), (l_0, \nu'_{0,0}), (l'_1, \nu_{0,1}), .., (l'_{n-1}, \nu_{0,n-1}), (l'_0, \nu_{1,0}), (l'_0, \nu'_{1,0}), ..,$$
$$(l''_{n-1}, \nu'_{1,n-1}), (l_0, \nu_{2,0}))$$

where $\nu_{0,0} = \nu_0$ and $\nu_{2,0} = \nu_q$. This subsequence results from exactly one execution of the appended cycle.

In the proof of lemma 5.3 we explained how to choose the minimal delays in the locations of the acceleratable cycle. Without proof, we apply the same technique here, with the exception that we choose the delay in the location l'_0 as the minimal delay plus $D - 2a$. This delay is larger or equal to the minimal delay, and thus assures that the guard on the edge to l''_1 is enabled. The invariant of l'_0 certainly allows this delay, since it is **true** by definition

5.5. To prove that this delay is also allowed by the invariant on location l_1'', we need our assumption that edge e_0 resets clock y. Since if that is the case, clock y equals zero on entry of location l_1'' and the invariant is certainly satisfied. As a result, the total delay of the subsequence is $a+D-2a+a = D$.

$\square$